Challenges in Compiling: Past, Present, and Future

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

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

September 2021
Outline

• What is a compiler?
• The biggest challenge: The Birth of a compiler
• The structure of modern compilers
• Modern challenges
• Conclusions
What is a Compiler?
Implementation Mechanisms

Source Program

Translator

Target Program

Machine
Implementation Mechanisms

Source Program → Translator → Target Program → Machine

Input Data

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
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

Program Specification

Machine
Implementation Mechanisms as “Bridges”

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

Program Specification

Translation

Lowering the level of specification

Machine
Implementation Mechanisms as “Bridges”

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

![Diagram](image_url)

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

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
Implementation Mechanisms as “Bridges”

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

Program Specification

Translation

Lowering the level of specification

Interpretation

Raising the level of execution

Machine

State: Variables, Operations: Assignments, Control Flow

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)
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 f8 48 8d 45 ec 48 89
48 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
48 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 8d 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 dd e8 1c
fe ff ff b8 00 00 00 00 48 8b 4d f8 64 48 33 0c 25 28 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 4f fd ff ff
48 8d 15 f5 2c 00 00 48 8d 35 5f 2f 00 00 48 8b 05 07 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 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 f8 48 c1 fd 03 74 1f 31 db
0f 1f 80 00 00 00 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]=0c218bf025a20bc43339dfe15cec41adc1c13946, for GNU/Linux 3.2.0, stripped
  ```

- `objdump -d a.out` produces the hexadecimal form along with assembly program
Why Is Compiler Construction a Relevant Subject?

Very few people write compilers any way
Why Is Compiler Construction a Relevant Subject?

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 Subject?

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 layers of abstractions and bridging the gaps between successive layers
Why Is Compiler Construction a Relevant Subject?

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 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 Subject?

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 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
The Birth of a Compiler
The First Compiler and “Real” Programming Language

- Fortran (later FORTRAN): 1956, Compiler: 1957
- Machine: IBM 704
- Creator: John Backus
  Richard Goldberg, Sheldon F. Best, Harlan Herrick, Peter Sheridan, Roy Nutt, Robert Nelson, Irving Ziller, Harold Stern, Lois Haibt, and David Sayre
The Beauty and The Beast

The Beauty and The Beast

Zuse (Plankalkul, 1945)
Curry (Composition, 1948)
Rutishauser (1951)
Bohm (1951)
Glennie (AUTOCODE, 1952)
Laning/Zierler (1953)
Hopper et al. (A-2, 1953)
Ershov (P.P., 1955)
Blum (ADES, 1956)
Perlis et al. (IT, 1956)
Mauchly et al. (Short Code, 1950)
Burks (Intermediate PL, 1950)
Goldstine/von Neumann (Flow Diagrams, 1946)
Brooker (Mark I Autocode, 1954)
Kamynin/Liubimskii (P.P., 19654)
Grems/Porter (Bacaic, 1955)
Elsworth et al. (Kompiler 2, 1955)
Katz et al. (MATH-MATIC, 1956-1958)
Hopper et al. (FLOW-MATIC, 1956-1958)
Bauer/Samelson (1956-1958)
Pioneers of Programming Languages (Knuth-Pardo, 1976)

Zuse (Plankalkul, 1945)
Curry (Composition, 1948)
Rutishauser (1951)
Bohm (1951)
Glennie (AUTOCODE, 1952)
Laning/Zierler (1953)
Hopper et al. (A-2, 1953)
Ershov (P.P., 1955)
Blum (ADES, 1956)
Perlis et al. (IT, 1956)

Mauchly et al. (Short Code, 1950)
Burks (Intermediate PL, 1950)
Goldstine/von Neumann (Flow Diagrams, 1946)
Brooker (Mark I Autocode, 1954)
Kamynin/Liubimskii (P.P., 19654)
Grems/Porter (Bacaic, 1955)
Elsworth et al. (Kompiler 2, 1955)
Katz et al. (MATH-MATIC, 1956-1958)
Hopper et al. (FLOW-MATIC, 1956-1958)
Bauer/Samelson (1956-1958)
Pioneers of Programming Languages (Knuth-Pardo, 1976)

- Zuse (Plankalkul, 1945)
- Curry (Composition, 1948)
- Rutishauser (1951)
- Bohm (1951)
- Glennie (AUTOCODE, 1952)
- Laning/Zierler (1953)
- Hopper et al. (A-2, 1953)
- Ershov (P.P., 1955)
- Blum (ADES, 1956)
- Perlis et al. (IT, 1956)

<table>
<thead>
<tr>
<th></th>
<th>Mauchly et al. (Short Code, 1950)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Burks (Intermediate PL, 1950)</td>
</tr>
<tr>
<td></td>
<td>Goldstine/von Neumann (Flow Diagrams, 1946)</td>
</tr>
<tr>
<td></td>
<td>Brooker (Mark I Autocode, 1954)</td>
</tr>
<tr>
<td></td>
<td>Kamynin/Liubimskii (P.P., 19654)</td>
</tr>
<tr>
<td></td>
<td>Grems/Porter (Bacaic, 1955)</td>
</tr>
<tr>
<td></td>
<td>Elsworth et al. (Kompiler 2, 1955)</td>
</tr>
<tr>
<td></td>
<td>Katz et al. (MATH-MATIC, 1956-1958)</td>
</tr>
<tr>
<td></td>
<td>Hopper et al. (FLOW-MATIC, 1956-1958)</td>
</tr>
<tr>
<td></td>
<td>Bauer/Samelson (1956-1958)</td>
</tr>
</tbody>
</table>

- Many efforts, and yet a breakthrough had to wait for Backus and his team
- We need to go back into the history to understand why it was so
Computing: Hand to Hand Combat with Machine (1)

• Computing was a black art

• Things available:
  The problem, the machine, the manual, and individual creativity

  “Computers were pretty crazy things. They had very primitive instructions and extremely bizarre input-output facilities.”

• Example: Selective Sequence Electronic Calculator (SSEC), 1948 - 1952
  Store of 150 words, Vacuum tubes and electro-mechanical relays
Computing: Hand to Hand Combat with Machine (2)

- No tools, only memory maps:
  - Machine Program in 0’s and 1’s + Data
  - Actual feeding by flipping switches
Computing: Hand to Hand Combat with Machine (2)

- No tools, only memory maps:
  - Machine Program in 0’s and 1’s + Data
  - Actual feeding by flipping switches

- **Assembler**:
  - Mnemonics + Symbolic references of addresses

- *(Absolute)* Loader:
  - read program from input device
  - enter program in appropriate memory locations
  - transfer control to the program
Computing: Hand to Hand Combat with Machine (2)

- No tools, only memory maps:
  - Machine Program in 0’s and 1’s + Data
  - Actual feeding by flipping switches

- **Assembler**:
  - Mnemonics + Symbolic references of addresses

  *(Absolute) Loader*:
  - read program from input device
  - enter program in appropriate memory locations
  - transfer control to the program

- **Macro-processor/Macro-assembler**
  - Combining many instructions for repeated use
Computing: Hand to Hand Combat with Machine (3)

- The story of paper tape
  - Punched paper tape glued to form a paper loop
  - Problem would appear and then disappear
  - Pattern repeated many times
  - Mobius strip

(Image source: Wikipedia)

- Debugging by the ear. When IBM 701 Defence Calculator arrived "How are we going to debug this enormous silent monster"
Beliefs of the Times

• Popular Mechanics Prediction in 1949
  *Computers in the future may weigh no more than 1.5 tons*
  
  (ENIAC, completed in 1947 weighed almost 30 tons)

• Editor of Prentice Hall business books, 1957
  *I have travelled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won’t last out the year*
Octal Humour

• “Why can’t programmers tell the difference between Christmas and New Year’s Eve? Because 25 in decimal is 31 in octal.”

• “We programmed it in octal. Thinking I was still a mathematician, I taught myself to add, subtract, and multiply, and even divide in octal. I was really good, until the end of the month, and then my check book didn’t balance! It stayed out of balance for three months until I got hold of my brother who was a banker. After several evenings of work he informed me that at intervals I had subtracted in octal. And I faced the major problem of living in two different worlds.”

  “That may have been one of the things that sent me to get rid of octal as far as possible.”

  — Grace Hopper
The Priesthood of Computing

- “Programming in the America of the 1950s had a vital frontier enthusiasm virtually untainted by either the scholarship or the stuffiness of academia.”

- “Programmer inventors of the early 1950s were too impatient to hoard an idea until it could be fully developed and a paper written. They wanted to convince others. Action, progress, and outdoing one’s rivals were more important than mere authorship of a paper.”

- “An idea was the property of anyone who could use it and the scholarly practice of noting references to sources and related work was almost universally unknown or unpractised.”
Obstacles in Creation of a High Level Language

- Priesthood wanted to preserve the order

“Priesthood wanted and got simple mechanical aids for the clerical drudgery which burdened them, but they regarded with hostility and derision more ambitious plans to make programming accessible to a larger population. To them, it was obviously a foolish and arrogant dream to imagine that any mechanical process could possibly perform the mysterious feats of invention required to write an efficient program.”
• Priesthood wanted to preserve the order

   “Priesthood wanted and got simple mechanical aids for the clerical drudgery which burdened them, but they regarded with hostility and derision more ambitious plans to make programming accessible to a larger population. To them, it was obviously a foolish and arrogant dream to imagine that any mechanical process could possibly perform the mysterious feats of invention required to write an efficient program.”

• There also were purveyors of snake oil

   “The energetic public relations efforts of some visionaries spread the word that their “automatic programming” systems had almost human abilities to understand the language and needs of the user; whereas closer inspection of these same systems would often reveal a complex, exception-ridden performer of clerical tasks which was both difficult to use and inefficient.”
The A2 Compiler

- Adding instructions to the machine viz. floating point operations
  - Programmers had a library of subroutine
  - They needed to copy the subroutine on the coding sheets by hand and change addresses manually

- Grace Hopper added a “call” operation whereby
  - the machine would copy the code
  - and update the addresses
The A2 Compiler

• Adding instructions to the machine viz. floating point operations
  ○ Programmers had a library of subroutine
  ○ They needed to copy the subroutine on the coding sheets by hand and change addresses manually

• Grace Hopper added a “call” operation whereby
  ○ the machine would copy the code Later called a linker
  ○ and update the addresses Later called a relocatable loader
The A2 Compiler

- Adding instructions to the machine viz. floating point operations
  - Programmers had a library of subroutine
  - They needed to copy the subroutine on the coding sheets by hand and change addresses manually
- Grace Hopper added a “call” operation whereby
  - the machine would copy the code
  - and update the addresses

The name “Compiler” was used because it put together a set of subroutines
The “Real” High Level Languages

• Conrad Zuse’s Plankalkul developed in a small village in Germany (1945)
  o “Program Calculus”
  o Only design, no implementation
    (Computers were destroyed in world war II)

• Laning and Zierler’s language for the WHIRLWIND at MIT (1953)
  o Fully algebraic in terms of supporting expressions
  o Very inefficient
Challenges for Creation of High Level Languages

- The tyranny of OR
  Expressiveness OR Efficiency
- Expressiveness:
  Higher level abstraction, features not supported by hardware
- Most time was spent in floating point subroutines
  - Not much attention was paid to address calculation, good use of registers
Challenges for Creation of High Level Languages

- The tyranny of OR
  Expressiveness OR Efficiency
- Expressiveness:
  Higher level abstraction, features not supported by hardware
- Most time was spent in floating point subroutines
  - Not much attention was paid to address calculation, good use of registers
- IBM 704 directly supported fast floating point operations
  - The need of expressiveness vanished revealing inefficiencies
    - Clumsy treatment of loops, indexing, references to registers
  - Led to rejection of “automatic programming”
The Genius of John Backus

He made the following important observations

- The main reason of inefficiency was a clumsy treatment of loops and array address computations
  If that could be handled, things may be far different
- The possibility made a lot of economic sense
- Language implementation was far more critical than language design
The Genius of John Backus

He made the following important observations

- The main reason of inefficiency was a clumsy treatment of loops and array address computations
  If that could be handled, things may be far different
- The possibility made a lot of economic sense
- Language implementation was far more critical than language design

_The “TRAN” in “FORTRAN” conveys the spirit_
The Genesis of FORTRAN

• Motivation:
  Programming and debugging costs already exceeded the cost of running a program, and as computers became faster and cheaper this imbalance would become more and more intolerable

• Goals: Can a machine translate
  - a sufficiently rich mathematical language into
  - a sufficiently economical program at
  - a sufficiently low cost

to make the whole affair feasible?

*The generated programs needed to be comparable to hand coded programs in efficiency*
• About Language Design
  o “We simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler that could produce efficient programs.”
  o “We had notions of assignment statements, subscripted variables, and the DO statement as the main features. Whatever else was needed emerged as we tried to build a way of programming on these basic ideas.”
The Design Philosophy

• About Language Design
  o “We simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler that could produce efficient programs.”
  o “We had notions of assignment statements, subscripted variables, and the DO statement as the main features. Whatever else was needed emerged as we tried to build a way of programming on these basic ideas.”

• About Compiler Design
  o Study the inner loops to find the most efficient method of execution
  o Find how the efficient code can be generated for sample statements
  o Generalize the observations by removing specificities and exceptions
The Design Philosophy

• About Language Design
  
  o "We simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler that could produce efficient programs."
  
  o "We had notions of assignment statements, subscripted variables, and the DO statement as the main features. Whatever else was needed emerged as we tried to build a way of programming on these basic ideas."

• About Compiler Design
  
  o Study the inner loops to find the most efficient method of execution
  
  o Find how the efficient code can be generated for sample statements
  
  o Generalize the observations by removing specificities and exceptions

Effectively, they raised the level of computing from

number processing to processing text that processed numbers
The FORTRAN Project

• Approved in Jan 1954, system delivered in April 1957
• Supportive management
• Young, energetic, enthusiastic, and inexperienced team
  ◦ Great team spirit and synergy
The FORTRAN Project

- Approved in Jan 1954, system delivered in April 1957
- Supportive management
- Young, energetic, enthusiastic, and inexperienced team
  - Great team spirit and synergy

“The best part was the uncertainty and excitement of waiting to see what kinds of object code all that work was finally going to produce.”
The FORTRAN Project

- Approved in Jan 1954, system delivered in April 1957
- Supportive management
- Young, energetic, enthusiastic, and inexperienced team
  - Great team spirit and synergy

"The best part was the uncertainty and excitement of waiting to see what kinds of object code all that work was finally going to produce."

"It was great sport in those days to scan the object program and either marvel at the translator or question its sanity!"
The FORTRAN Project

- Approved in Jan 1954, system delivered in April 1957
- Supportive management
- Young, energetic, enthusiastic, and inexperienced team
  - Great team spirit and synergy
    - "The best part was the uncertainty and excitement of waiting to see what kinds of object code all that work was finally going to produce."
    - "It was great sport in those days to scan the object program and either marvel at the translator or question its sanity!"
  - Helped in ignoring the doubters and overcome discouragement and despair
• “The amount of knowledge necessary to utilize the 704 effectively by means of FORTRAN is far less than the knowledge required to make effective use of the 704 by direct coding. It will be possible to make the full capabilities of the 704 available to a much wider range of people than would otherwise be possible without expensive and time-consuming training programs.”
FORTRAN Claims (1)

• “The amount of knowledge necessary to utilize the 704 effectively by means of FORTRAN is far less than the knowledge required to make effective use of the 704 by direct coding.

   It will be possible to make the full capabilities of the 704 available to a much wider range of people than would otherwise be possible without expensive and time-consuming training programs.”

• “FORTRAN may apply complex, lengthy techniques in coding a problem which the human coder would have neither the time nor inclination to derive or apply.”
FORTRAN Claims (1)

- “The amount of knowledge necessary to utilize the 704 effectively by means of FORTRAN is far less than the knowledge required to make effective use of the 704 by direct coding. It will be possible to make the full capabilities of the 704 available to a much wider range of people than would otherwise be possible without expensive and time-consuming training programs.”

- FORTRAN may apply complex, lengthy techniques in coding a problem which the human coder would have neither the time nor inclination to derive or apply.”

- Replace IBM 704 by your favorite multi-core processor

- Replace “complex lengthy technique” by scheduling for parallel computing

- Imagine a language for it
“FORTRAN will virtually eliminate coding and debugging” 😊
FORTRAN Claims (2)

“FORTRAN will virtually eliminate coding and debugging” 😊
Limitations of FORTRAN I Language

• No reserved words
• Tokenization ignored spaces
• Simplistic functions
• No subprograms, no recursion
• No spaces
• DO loops with limited nesting depth of 3
• Implicit types based on the first letter
• No declarations required
Minor Errors Could be Rather Expensive

- The first American Venus probe was lost because of a computer problem
- A programmer replaced a comma by a dot

<table>
<thead>
<tr>
<th>Should have been</th>
<th>Was</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO 10 I = 1, 3</td>
<td>DO 10 I = 1. 3</td>
</tr>
</tbody>
</table>

- What was essentially a DO loop header got treated as an assignment statement DO10I = 1.3 by the compiler
Fun with FORTRAN

- Implicit types based on the first letter
  - I, J, K, L, M, N: Integer
  - Others: Real

- No reserved words
Fun with FORTRAN

- Implicit types based on the first letter
  - I,J,K,L,M,N: Integer
  - Others: Real

  "GOD is real unless declared integer".

- No reserved words

  IF (IF .LT. THEN) THEN ELSE = THEN ELSE THEN = ELSE
Contributions of FORTRAN I Compiler

- Phase-wise division of work
- Optimizations:
  - Common subexpressions elimination,
  - Array address optimization in loops
    (a form of strength reduction and induction variable elimination)
  - Register allocation using hierarchical regions
    (optimal under number of loads for straight line code)
- Basic blocks and execution frequency analysis
- Distinction between pseudo registers and hard registers
Expressions in the Programs

- Other “algebraic” compilers needed parenthesis for expressions
- No concept for parsing using grammars

<table>
<thead>
<tr>
<th>Expression</th>
<th>Expression Tree</th>
<th>Required Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a + b^{<em>}c^{</em>}d + e$</td>
<td><img src="image" alt="Expression Tree" /></td>
<td>$(a + ((b^{<em>}c^{</em>}d + e))$</td>
</tr>
</tbody>
</table>
FORTRAN Rules for Expressions

1. Any fixed point (floating point) constant, variable, or subscripted variable is an expression of the same mode. Thus 3 and \(I\) are fixed point expressions, and \(ALPHA\) and \(A(I, J, K)\) are floating point expressions.

2. If \(SOMEF\) is some function of \(n\) variables, and if \(E, F, \ldots, H\) are a set of \(n\) expressions of the correct modes for \(SOMEF\), then \(SOMEF(E, F, \ldots, H)\) is an expression of the same mode as \(SOMEF\).

3. If \(E\) is an expression, and if its first character is not “+” or “−”, then \(+E\) and \(-E\) are expressions of the same mode as \(E\). Thus \(-A\) is an expression, but \(- - A\) is not.

4. If \(E\) is an expression, then \((E)\) is an expression of the same mode as \(E\). Thus \((A), ((A)), (((A)))\), etc. are expressions.

5. If \(E\) and \(F\) are expressions of the same mode, and if the first character of \(F\) is not + or −, then \(E + F\), \(E − F\), \(E \times F\), \(E/F\) are expressions of the same mode.
• Conventional precedences were used and parenthesis were not required.

• Simple rule of reconstructing parenthesized expressions:
  Assuming three levels of precedences of “+”, “∗”, and “∗∗”
  
  o Add “(((” in the beginning of the expression
    (and hence before every “(“ in the expression)
  o Add “))))” at the end of the expression
    (and hence after every “)” in the expression)
  o Replace every “+” by “)))) + (((“
  o Replace every “∗” by “)) ∗ (((“
  o Replace every “∗∗” by “)) ∗ ∗(“
FORTRAN Expression Handling

- Conventional precedences were used and parenthesis were not required.
- Simple rule of reconstructing parenthesized expressions:
  Assuming three levels of precedences of “+”, “*”, and “**”
  
  - Add “((“ in the beginning of the expression
    (and hence before every “(“ in the expression)
  - Add “)))” at the end of the expression
    (and hence after every “)” in the expression)
  - Replace every “+” by “))) + (((”
  - Replace every “*” by “))) * (((”
  - Replace every “**” by “))) * * (”
  
  - Our expression becomes fully parenthesized by application of this rule.

\[ A + B ** C * (D + E) \]
FORTRAN Expression Handling

- Conventional precedences were used and parenthesis were not required.
- Simple rule of reconstructing parenthesized expressions:
  Assuming three levels of precedences of “+”, “*”, and “**”
  - Add “((((” in the beginning of the expression
    (and hence before every “(“ in the expression)
  - Add “))))” at the end of the expression
    (and hence after every “)” in the expression)
  - Replace every “+” by “))) + (((”
  - Replace every “*” by “)) * ((”
  - Replace every “**” by “)) * *(”

- Our expression becomes fully parenthesized by application of this rule.

\[
(((A + \quad B \quad ** \quad C \quad * \quad (D \quad + \quad E))\]

34/83
**FORTRAN Expression Handling**

- Conventional precedences were used and parenthesis were not required.
- Simple rule of reconstructing parenthesized expressions:
  
  Assuming three levels of precedences of “+”, “*”, and “**”

  - Add “((“ in the beginning of the expression (and hence before every “(“ in the expression)
  - Add “)))” at the end of the expression (and hence after every “)“ in the expression)
  - Replace every “+” by “))” + “))”
  - Replace every “*” by “))” * “))”
  - Replace every “**” by “))” * “))”

- Our expression becomes fully parenthesized by application of this rule.

\[
(((A + B \ast \ast C) \ast (((D + E)))
\]
FORTRAN Expression Handling

- Conventional precedences were used and parenthesis were not required.
- Simple rule of reconstructing parenthesized expressions:
  Assuming three levels of precedences of “+”, “∗”, and “∗∗”
  - Add “((“ in the beginning of the expression
    (and hence before every “(" in the expression)
  - Add “)))” at the end of the expression
    (and hence after every “)" in the expression)
  - Replace every “+” by “))” + “((“
  - Replace every “∗” by “))” * “((“
  - Replace every “∗∗” by “)) * *(“

- Our expression becomes fully parenthesized by application of this rule.

\[
(((A + B * * C * (((D + E) )))
\]
FORTRAN Expression Handling

- Conventional precedences were used and parenthesis were not required.
- Simple rule of reconstructing parenthesized expressions:
  Assuming three levels of precedences of “+”, “∗”, and “∗∗”
  - Add “((“ in the beginning of the expression
    (and hence before every “(" in the expression)
  - Add “))” at the end of the expression
    (and hence after every “)" in the expression)
  - Replace every “+” by “))” + “)("
  - Replace every “∗” by “))” ∗ “)("
  - Replace every “∗∗” by “))” ∗ ∗ “)("

- Our expression becomes fully parenthesized by application of this rule.

\(((A + B ∗∗ C ) ∗ (((D + E)))))))\)
FORTRAN Expression Handling

• Conventional precedences were used and parenthesis were not required.

• Simple rule of reconstructing parenthesized expressions:
  Assuming three levels of precedences of “+”, “*”, and “**”
  
  o Add “(((“ in the beginning of the expression
    (and hence before every “(“ in the expression)
  o Add “))))” at the end of the expression
    (and hence after every “)” in the expression)
  o Replace every “+” by “)))+ (((“
  o Replace every “*” by “)))* ( (“
  o Replace every “**” by “)))* *(“

• Our expression becomes fully parenthesized by application of this rule.

  (((((A)))) + (((B ** C  * (((D))) + (((E)))))))
FORTRAN Expression Handling

- Conventional precedences were used and parenthesis were not required.
- Simple rule of reconstructing parenthesized expressions:
  
  Assuming three levels of precedences of “+”, “*”, and “**”
  
  - Add “((“ in the beginning of the expression
    (and hence before every “(“ in the expression)
  - Add “))” at the end of the expression
    (and hence after every “)” in the expression)
  - Replace every “+” by “))” + “((”
  - Replace every “*” by “)) * “((”
  - Replace every “**” by “)) * **)”

- Our expression becomes fully parenthesized by application of this rule.

  (((((A))) + ((((B) ** C)) * ((((((D))) + (((E)))))))))
FORTRAN Expression Handling

- Conventional precedences were used and parenthesis were not required.
- Simple rule of reconstructing parenthesized expressions:
  Assuming three levels of precedences of “+”, “∗”, and “∗∗”
  - Add “((“ in the beginning of the expression (and hence before every “(" in the expression)
  - Add “)))” at the end of the expression (and hence after every “")” in the expression)
  - Replace every “+” by “))” + “(“
  - Replace every “∗” by “)∗ ) (∗“)
  - Replace every “∗∗” by “)∗ ∗(""

- Our expression becomes fully parenthesized by application of this rule.

\[ (((A)) + (((B) ∗ ∗(C)) ∗ (((((D)) + (((E)))))))) \]
FORTRAN Expression Handling

- Conventional precedences were used and parenthesis were not required.
- Simple rule of reconstructing parenthesized expressions:
  Assuming three levels of precedences of “+”, “*”, and “**”
  - Add “((“ in the beginning of the expression
    (and hence before every “(“ in the expression)
  - Add “))” at the end of the expression
    (and hence after every “)” in the expression)
  - Replace every “+” by “))” + (”
  - Replace every “*” by “)) * (“
  - Replace every “**” by “)) * *(“

- Our expression becomes fully parenthesized by application of this rule.

```
(((A))) + (((B) **(C)) * ((((((D)) + (((E)))))
```

(The rules can be applied in a single left-to-right scan of the expression)
FORTRAN Compiler Anecdotes (1)

- Expression computation problem observed by Bernard A. Galler
  - For $n = 10$, the expression $n \times (n - 1)/2$ computed 40 instead of 45!
• Expression computation problem observed by Bernard A. Galler
  
  o For \( n = 10 \), the expression \( n \times (n - 1)/2 \) computed 40 instead of 45!
  
  o “/” had a higher precedence and 9/2 is 4 in integer arithmetic
FORTRAN Compiler Anecdotes (1)

• Expression computation problem observed by Bernard A. Galler
  o For \( n = 10 \), the expression \( n \times (n - 1)/2 \) computed 40 instead of 45!
  o “/” had a higher precedence and 9/2 is 4 in integer arithmetic

• Response from IBM
  “It is too complicated to change the compiler so we will fix the manual”
FORTRAN Compiler Anecdotes (1)

- Expression computation problem observed by Bernard A. Galler
  - For $n = 10$, the expression $n \times (n - 1)/2$ computed 40 instead of 45!
  - "/" had a higher precedence and $9/2$ is 4 in integer arithmetic

- Response from IBM
  "It is too complicated to change the compiler so we will fix the manual"

- New manual had the following statement:
  "Please be warned that mathematical equivalence is not the same as computational equivalence"
• Expression computation problem observed by Bernard A. Galler
  ○ For \( n = 10 \), the expression \( n \times (n - 1)/2 \) computed 40 instead of 45!
  ○ "/" had a higher precedence and 9/2 is 4 in integer arithmetic

• Response from IBM
  "It is too complicated to change the compiler so we will fix the manual"

• New manual had the following statement:
  "Please be warned that mathematical equivalence is not the same as computational equivalence"

• How about the same precedence for "/" and "*" and left associativity?
FORTRAN Compiler Anecdotes (1)

• Expression computation problem observed by Bernard A. Galler
  - For $n = 10$, the expression $n \times (n - 1)/2$ computed 40 instead of 45!
  - "/" had a higher precedence and $9/2$ is 4 in integer arithmetic

• Response from IBM
  "It is too complicated to change the compiler so we will fix the manual"

• New manual had the following statement:
  "Please be warned that mathematical equivalence is not the same as computational equivalence"

• How about the same precedence for "/" and "\times" and left associativity?
  - $n/2 \times (n - 1)$
  - $n \times (n - 1) \times (1/2)$
On compiler reliability

- Tables stored on the magnetic drum based memory
- Slow searches and more load on drums
- The compiler worked far better at GM than at Westinghouse
- GM people had ensured a much better servicing of magnetic drums!
ON compiler efficiency

- Frank Engel at Westinghouse observed that tapes moved independently but sequentially
- Compiler could become faster if tape movement is made to overlap
- Frank asked for the source and got a reply: (source meant assembly)
  “IBM does not supply source code”
- Frank patched up the octal object code of the compiler and the throughput increased by a factor of 3!
- IBM was surprised and wanted a copy, so Frank said:
  “Westinghouse does not supply object code”
A FORTRAN Program for Array Copy

Program

DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
DO 1 I = 1, 10
1 A(I,J) = B(I,J)
A FORTRAN Program for Array Copy

**Program**

```fortran
DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
    DO 1 I = 1, 10
        1 A(I,J) = B(I,J)
```

**A simplified view for 4x3 fragments**

<table>
<thead>
<tr>
<th></th>
<th>B(1,1)</th>
<th>B(1,2)</th>
<th>B(1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(2,1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(2,2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(2,3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(3,1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(3,2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(3,3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(4,1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(4,2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(4,3)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A(1,1)</th>
<th>A(1,2)</th>
<th>A(1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(2,1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(2,2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(2,3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(3,1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(3,2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(3,3)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(4,1)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(4,2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A(4,3)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A FORTRAN Program for Array Copy

Program

```
DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
    1 A(I,J) = B(I,J)
```

A simplified view for 4x3 fragments

```

B(1,1) B(1,2) B(1,3)
B(2,1) B(2,2) B(2,3)
B(3,1) B(3,2) B(3,3)
B(4,1) B(4,2) B(4,3)

A(1,1) A(1,2) A(1,3)
A(2,1) A(2,2) A(2,3)
A(3,1) A(3,2) A(3,3)
A(4,1) A(4,2) A(4,3)
```
A FORTRAN Program for Array Copy

<table>
<thead>
<tr>
<th>DIMENSION A (10,10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIMENSION B (10,10)</td>
</tr>
<tr>
<td>DO 1 J = 1, 10</td>
</tr>
<tr>
<td>DO 1 I = 1, 10</td>
</tr>
<tr>
<td>1 A(I,J) = B(I,J)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A simplified view for 4x3 fragments</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1,1)</td>
</tr>
<tr>
<td>B(2,1)</td>
</tr>
<tr>
<td>B(3,1)</td>
</tr>
<tr>
<td>B(4,1)</td>
</tr>
<tr>
<td>A(1,1)</td>
</tr>
<tr>
<td>A(2,1)</td>
</tr>
<tr>
<td>A(3,1)</td>
</tr>
<tr>
<td>A(4,1)</td>
</tr>
</tbody>
</table>
A FORTRAN Program for Array Copy

**Program**

```fortran
DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
    1 A(I,J) = B(I,J)
```

**A simplified view for 4x3 fragments**

<table>
<thead>
<tr>
<th>B(1,1)</th>
<th>B(1,2)</th>
<th>B(1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(2,1)</td>
<td>B(2,2)</td>
<td>B(2,3)</td>
</tr>
<tr>
<td>B(3,1)</td>
<td>B(3,2)</td>
<td>B(3,3)</td>
</tr>
<tr>
<td>B(4,1)</td>
<td>B(4,2)</td>
<td>B(4,3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(1,1)</th>
<th>A(1,2)</th>
<th>A(1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(2,1)</td>
<td>A(2,2)</td>
<td>A(2,3)</td>
</tr>
<tr>
<td>A(3,1)</td>
<td>A(3,2)</td>
<td>A(3,3)</td>
</tr>
<tr>
<td>A(4,1)</td>
<td>A(4,2)</td>
<td>A(4,3)</td>
</tr>
</tbody>
</table>
A FORTRAN Program for Array Copy

Program

```
DIMENSION A (10,10)  
DIMENSION B (10,10)  
DO 1 J = 1, 10  
  DO 1 I = 1, 10  
    1 A(I,J) = B(I,J)
```

A simplified view for 4x3 fragments

<table>
<thead>
<tr>
<th></th>
<th>B(1,1)</th>
<th>B(1,2)</th>
<th>B(1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(2,1)</td>
<td>B(2,2)</td>
<td>B(2,3)</td>
<td></td>
</tr>
<tr>
<td>B(3,1)</td>
<td>B(3,2)</td>
<td>B(3,3)</td>
<td></td>
</tr>
<tr>
<td>B(4,1)</td>
<td>B(4,2)</td>
<td>B(4,3)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A(1,1)</th>
<th>A(1,2)</th>
<th>A(1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(2,1)</td>
<td>A(2,2)</td>
<td>A(2,3)</td>
<td></td>
</tr>
<tr>
<td>A(3,1)</td>
<td>A(3,2)</td>
<td>A(3,3)</td>
<td></td>
</tr>
<tr>
<td>A(4,1)</td>
<td>A(4,2)</td>
<td>A(4,3)</td>
<td></td>
</tr>
</tbody>
</table>
**A FORTRAN Program for Array Copy**

**Program**

```fortran
DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
    1 A(I,J) = B(I,J)
```

---

**A simplified view for 4x3 fragments**

<table>
<thead>
<tr>
<th>B(1,1)</th>
<th>B(2,2)</th>
<th>B(3,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(2,1)</td>
<td>B(2,2)</td>
<td>B(2,3)</td>
</tr>
<tr>
<td>B(3,1)</td>
<td>B(3,2)</td>
<td>B(3,3)</td>
</tr>
<tr>
<td>B(4,1)</td>
<td>B(4,2)</td>
<td>B(4,3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(1,1)</th>
<th>A(2,2)</th>
<th>A(3,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(2,1)</td>
<td>A(2,2)</td>
<td>A(2,3)</td>
</tr>
<tr>
<td>A(3,1)</td>
<td>A(3,2)</td>
<td>A(3,3)</td>
</tr>
<tr>
<td>A(4,1)</td>
<td>A(4,2)</td>
<td>A(4,3)</td>
</tr>
</tbody>
</table>
A FORTRAN Program for Array Copy

```
DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
    1 A(I,J) = B(I,J)
```

A simplified view for 4x3 fragments

```
<table>
<thead>
<tr>
<th>B(1,1)</th>
<th>B(1,2)</th>
<th>B(1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(2,1)</td>
<td>B(2,2)</td>
<td>B(2,3)</td>
</tr>
<tr>
<td>B(3,1)</td>
<td>B(3,2)</td>
<td>B(3,3)</td>
</tr>
<tr>
<td>B(4,1)</td>
<td>B(4,2)</td>
<td>B(4,3)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A(1,1)</th>
<th>A(1,2)</th>
<th>A(1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(2,1)</td>
<td>A(2,2)</td>
<td>A(2,3)</td>
</tr>
<tr>
<td>A(3,1)</td>
<td>A(3,2)</td>
<td>A(3,3)</td>
</tr>
<tr>
<td>A(4,1)</td>
<td>A(4,2)</td>
<td>A(4,3)</td>
</tr>
</tbody>
</table>
```
A FORTRAN Program for Array Copy

Program

```
DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
    1  A(I,J) = B(I,J)
```

A simplified view for 4x3 fragments

```
B(1,1) | B(1,2) | B(1,3)  
B(2,1) | B(2,2) | B(2,3)  
B(3,1) | B(3,2) | B(3,3)  
B(4,1) | B(4,2) | B(4,3)  
```

```
A(1,1) | A(1,2) | A(1,3)  
A(2,1) | A(2,2) | A(2,3)  
A(3,1) | A(3,2) | A(3,3)  
A(4,1) | A(4,2) | A(4,3)  
```
A FORTRAN Program for Array Copy

Program

```
DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
    1 A(I,J) = B(I,J)
```

A simplified view for 4x3 fragments

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>B2</td>
</tr>
<tr>
<td>B1</td>
<td>B2</td>
</tr>
<tr>
<td>B1</td>
<td>B2</td>
</tr>
</tbody>
</table>
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>A2</td>
</tr>
<tr>
<td>A1</td>
<td>A2</td>
</tr>
<tr>
<td>A1</td>
<td>A2</td>
</tr>
</tbody>
</table>
```

A FORTRAN Program for Array Copy

Program

DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
    1 A(I,J) = B(I,J)

A simplified view for 4x3 fragments

<table>
<thead>
<tr>
<th></th>
<th>B(1,1)</th>
<th>B(2,1)</th>
<th>B(3,1)</th>
<th>B(4,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(1,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(1,3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(2,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(2,3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(3,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(3,3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(4,2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B(4,3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### A FORTRAN Program for Array Copy

**Program**

```fortran
DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
    1 A(I,J) = B(I,J)
```

**A simplified view for 4x3 fragments**

<table>
<thead>
<tr>
<th></th>
<th>B(1,1)</th>
<th>B(1,2)</th>
<th>B(1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>B(2,1)</td>
<td>B(2,2)</td>
<td>B(2,3)</td>
<td></td>
</tr>
<tr>
<td>B(3,1)</td>
<td>B(3,2)</td>
<td>B(3,3)</td>
<td></td>
</tr>
<tr>
<td>B(4,1)</td>
<td>B(4,2)</td>
<td>B(4,3)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>A(1,1)</th>
<th>A(1,2)</th>
<th>A(1,3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(2,1)</td>
<td>A(2,2)</td>
<td>A(2,3)</td>
<td></td>
</tr>
<tr>
<td>A(3,1)</td>
<td>A(3,2)</td>
<td>A(3,3)</td>
<td></td>
</tr>
<tr>
<td>A(4,1)</td>
<td>A(4,2)</td>
<td>A(4,3)</td>
<td></td>
</tr>
</tbody>
</table>
A FORTRAN Program for Array Copy

Program

DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
    1 A(I,J) = B(I,J)
A FORTRAN Program for Array Copy

Program

```
DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
    1 A(I,J) = B(I,J)
```

A simplified view for 4x3 fragments
Array Address Calculation

Cell \((i, j)\)

Its address

\[ x \]
Array Address Calculation

Cell \((i, j)\)

\[ \text{Its address} \]

\[ \text{Base} + (j - 1) \times 10 + i - 1 \]
Array Address Calculation

Cell \((i,j)\)

Its address

\[ \text{Base} + (j - 1) \times 10 + i - 1 \]

An additional complication: In FORTRAN, arrays are stored backwards and index registers are subtracted from the base
Output of FORTRAN I Compiler

### Source Program

```
DIMENSION A (10,10)
DIMENSION B (10,10)

DO 1 J = 1, 10
  DO 1 I = 1, 10
  1    A(I,J) = B(I,J)
```

### Object Program

<table>
<thead>
<tr>
<th>Statement</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LXD ONE, 1</td>
<td>$lrx1 = 1$</td>
</tr>
<tr>
<td>LOOP CLA B+1, 1</td>
<td>$Acc = *(B + 1 - lrx1)$</td>
</tr>
<tr>
<td>ST0 A+1, 1</td>
<td>$(A + 1 - lrx1) = Acc$</td>
</tr>
<tr>
<td>TXI * +1, 1, 1</td>
<td>$lrx1 = lrx1 + 1$, jump ahead by 1</td>
</tr>
<tr>
<td>TXL LOOP,1 ,100</td>
<td>if ($lrx1 \leq 100$), goto LOOP</td>
</tr>
</tbody>
</table>
Source Program

```
DIMENSION A (10,10)
DIMENSION B (10,10)
DO 1 J = 1, 10
DO 1 I = 1, 10
1 A(I,J) = B(I,J)
```

Object Program

<table>
<thead>
<tr>
<th>Statement</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LXD ONE, 1</td>
<td>lxr1 = 1</td>
</tr>
<tr>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td>CLA B+1, 1</td>
<td>Acc = *(B + 1 − lxr1)</td>
</tr>
<tr>
<td>ST0 A+1, 1</td>
<td>*(A + 1 − lxr1) = Acc</td>
</tr>
<tr>
<td>TXI * +1, 1, 1</td>
<td>lxr1 = lxr1 + 1, jump ahead by 1</td>
</tr>
<tr>
<td>TXL LOOP, 1, 100</td>
<td>if (lxr1 ≤ 100), goto LOOP</td>
</tr>
</tbody>
</table>
Compiling Array Copy Program: Control Flow Graph

**DIMENSION A (10,10)**
**DIMENSION B (10,10)**

DO 1 J = 1, 10
DO 1 I = 1, 10
1 A(I,J) = B(I,J)

\[ t_1 = (j - 1) \times 10 + i - 1 \]
\[ t_2 = (B - t_1) \times (A - t_1) = t_2 \]
\[ i = i + 1 \]

\((i > 10)\) \(\rightarrow\) \((i \leq 10)\)
\((j > 10)\) \(\rightarrow\) \((j \leq 10)\)
Compiling Array Copy Program: Strength Reduction (1)

Observations about the inner loop

\[ i = j = 1 \]

\[ t_1 = (j - 1) \times 10 + i - 1 \]
\[ t_2 = *(B - t_1) \]
\[ *(A - t_1) = t_2 \]
\[ i = i + 1 \]

\( (i > 10) \text{ (i ≤ 10)} \)
\( (j > 10) \text{ (j ≤ 10)} \)

\( j = j + 1 \)
Compiling Array Copy Program: Strength Reduction (1)

Observations about the inner loop
- Whenever i increments by 1, t1 also increments by 1

```
i = j = 1
```

```
t1 = (j - 1) * 10 + i - 1

01 = *(B - t1)

*(A - t1) = t2

i = i + 1
```

```
(i > 10)  (i ≤ 10)
```

```
j = j + 1
```

```
(j > 10)  (j ≤ 10)
```
Compiling Array Copy Program: Strength Reduction (1)

```
i = j = 1
```

```
t_1 = (j - 1) * 10 + i - 1
```

```
t_2 = *(B - t_1)
```

```
*(A - t_1) = t_2
```

```
i = i + 1
```

Observations about the inner loop

- Whenever $i$ increments by 1, $t_1$ also increments by 1
- We can initialize $t_1$ outside of the inner loop

```
t_1 = (j - 1) * 10 + i - 1
```

```
t_1 = (j - 1) * 10
```

(because $i$ is 1)

and increment it within the loop

```
t_1 = t_1 + 1
```
Compiling Array Copy Program: Strength Reduction (2)

\[
\begin{align*}
  i &= j = 1 \\
  t_1 &= (j - 1) \times 10 \\
  t_2 &= \ast(B - t_1) \\
  \ast(A - t_1) &= t_2 \\
  i &= i + 1 \\
  t_1 &= t_1 + 1 \\
  (i > 10) &\quad (i \leq 10) \\
  j &= j + 1 \\
  (j > 10) &\quad (j \leq 10)
\end{align*}
\]
Compiling Array Copy Program: Strength Reduction (2)

Observations about the inner loop

- Whenever \( j \) increments by 1, \( t1 \) increments by 10
Observations about the inner loop

- Whenever \( j \) increments by 1, \( t1 \) increments by 10
- We can initialize \( t1 \) outside of the outer loop
  
  \[
  t1 = (j - 1) \times 10
  \]
  
  and increment it within the loop
  
  \[
  t1 = t1 + 10
  \]
Compiling Array Copy Program: Strength Reduction (2)

Observations about the inner loop

- Whenever $j$ increments by 1, $t_1$ increments by 10.
- We can initialize $t_1$ outside of the outer loop.
  
  \[ t_1 = (j - 1) \times 10 \]
  \[ t_2 = (B - t_1) \times (A - t_1) = t_2 \]
  \[ i = i + 1 \]
  
  $t_1 = t_1 + 1$

  (because $j$ is 1)

  and increment it within the loop
  
  \[ t_1 = t_1 + 10 \]

  (because $j$ is 1)

- However, the inner loop already increments $t_1$ by 10.

\[ i = j = 1 \]

\[ t_1 = (j - 1) \times 10 \]

\[ t_2 = (B - t_1) \times (A - t_1) = t_2 \]

\[ i = i + 1 \]

\[ j = j + 1 \]

(i > 10)  (i ≤ 10)  (i > 10)  (i ≤ 10)
Compiling Array Copy Program: Flattening the Loops

\[
i = j = 1 \\
\text{ } \\
\text{ } \\
t_1 = 0
\]

\[
t_2 = *(B - t_1) \\
*(A - t_1) = t_2 \\
i = i + 1 \\
t_1 = t_1 + 1
\]

(i > 10)  (i ≤ 10)

(j > 10)  (j ≤ 10)
The only activity in the outer loop now is to control the loop iterations
No other computation

\begin{align*}
i &= j = 1 \\
t_1 &= 0 \\
\end{align*}

\begin{align*}
t_2 &= *(B - t_1) \\
*&(A - t_1) = t_2 \\
i &= i + 1 \\
t_1 &= t_1 + 1 \\
\end{align*}

\begin{align*}
(i > 10) &\quad (i \leq 10) \\
\end{align*}

\begin{align*}
j &= j + 1 \\
\end{align*}

\begin{align*}
(j > 10) &\quad (j \leq 10) \\
\end{align*}
Compiling Array Copy Program: Flattening the Loops

- The only activity in the outer loop now is to control the loop iterations
  No other computation
- We can combine the loops into a single loop by taking a product of the two loop bounds
- Variables $i$ and $j$ would not be required
Compiling Array Copy Program: The Final Program

Control flow graph (CFG)  |  Original Assembly

```
t1 = 0

\[ \begin{align*}
  t2 &= *(B - t1) \\
  *(A - t1) &= t2 \\
  t1 &= t1 + 1
\end{align*} \]
```

(t1 > 100)  |  (t1 \leq 100)
Compiling Array Copy Program: The Final Program

Control flow graph (CFG) | Original Assembly
--- | ---

$t1 = 0$

$t2 = *(B - t1)$

$*(A - t1) = t2$

$t1 = t1 + 1$

$(t1 > 100)$

$(t1 \leq 100)$

LXD ONE, 1

LOOP CLA B+1, 1

ST0 A+1, 1

TXI * +1, 1, 1

TXL LOOP,1,100

Original Assembly

```
  t1 = 0
  t2 = *(B - t1)
  *(A - t1) = t2
  t1 = t1 + 1

  (t1 > 100)
  (t1 \leq 100)
```

Uday Khedker
IIT Bombay

Talk Title: Compiling-Challenges

Topic:
Outline

What is a Compiler?

The Birth of a Compiler

The Structure of Modern Compilers

Modern Challenges

Conclusions
Compiling Array Copy Program: The Final Program

Control flow graph (CFG)  Original Assembly

\[ t1 = 0 \]

\[ t2 = (B - t1) \]
\[ *(A - t1) = t2 \]
\[ t1 = t1 + 1 \]

\( (t1 > 100) \)
\( (t1 \leq 100) \)

LXD ONE, 1
LOOP CLA B+1, 1
STO A+1, 1
TXI * +1, 1, 1
TXL LOOP,l ,100

<table>
<thead>
<tr>
<th>Minor differences</th>
<th>CFG</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base address</td>
<td>B</td>
<td>B+1</td>
</tr>
<tr>
<td>Initial value of t1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Compiling Array Copy Program Using GCC 4.7.2 (gfortran)

.L5:
  leal 408(%esp), %ebx
  movl $1, %eax
  leal 808(%esp), %ecx
  addl %esi, %ebx
  addl %esi, %ecx

.p2align 4,,7

.L4:
  movl -44(%ecx,%eax,4), %edx
  movl %edx, -44(%ebx,%eax,4)
  addl $1, %eax
  cmpl $11, %eax
  jne .L4
  addl $40, %esi
  cmpl $400, %esi
  jne .L5

• Integer is now 4 bytes
Compiling Array Copy Program Using GCC 4.7.2 (gfortran)

.L5:
leal 408(%esp), %ebx
movl $1, %eax
leal 808(%esp), %ecx
addl %esi, %ebx
addl %esi, %ecx
.p2align 4,,7
.p2align 3

.L4:
movl -44(%ecx,%eax,4), %edx
movl %edx, -44(%ebx,%eax,4)
addl $1, %eax
cmpl $11, %eax
jne .L4
addl $40, %esi
cmpl $400, %esi
jne .L5

- Integer is now 4 bytes
- Efficient address calculation with strength reduction
Compiling Array Copy Program Using GCC 4.7.2 (gfortran)

.L5:
leal 408(%esp), %ebx
movl $1, %eax
leal 808(%esp), %ecx
addl %esi, %ebx
addl %esi, %ecx
.p2align 4,,7
.p2align 3

.L4:
movl -44(%ecx,%eax,4), %edx
movl %edx, -44(%ebx,%eax,4)
addl $1, %eax
cmpl $11, %eax
jne .L4
addl $40, %esi
cmpl $400, %esi
jne .L5

- Integer is now 4 bytes
- Efficient address calculation with strength reduction
- Nested loops not flattened
The Structure of Modern Compilers
Language Implementation Models

- Analysis
- Synthesis
- Execution

- Compilation
- Interpretation

**Analysis**

**Synthesis**

**Execution**

**Talk Title:** Compiling-Challenges

**Topic:**
- Outline
- What is a Compiler?
- The Birth of a Compiler
- The Structure of Modern Compilers
- Modern Challenges
- Conclusions
Language Processor Models

- C, C++
- Java, C#
Reusability of Language Processor Modules

- Front Ends
  - Language 1
  - Language 2
  - Language $n$

- Common IR

- Back Ends
  - Machine 1
  - Machine 2
  - Machine $m$
m \times n \text{ compilers can be obtained from } m + n \text{ modules}
Compilation Models

Aho Ullman Model

Davidson Fraser Model

Uday Khedker
IIT Bombay

Talk Title:
Compiling-Challenges

Topic:
Outline

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
Compilation Models

Aho Ullman Model

Davidson Fraser Model

Input Source Program

Front End

AST

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
Outline:

- What is a Compiler?
- The Birth of a Compiler
- The Structure of Modern Compilers
- Modern Challenges
- Conclusions

Compilation Models

**Aho Ullman Model**

1. Input Source Program
2. Front End
3. AST
4. Optimizer
5. Target Indep. IR

**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

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
Talk Title: Compiling-Challenges

Topic: Outline

Outline:
- What is a Compiler?
- The Birth of a Compiler
- The Structure of Modern Compilers
- Modern Challenges
- Conclusions

Compilation Models

Aho Ullman Model

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

Davidson Fraser Model

Input Source Program → Front End → AST
Compilation Models

**Aho Ullman Model**

Input Source Program → 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

Input Source Program → Front End → AST → Expander → Register Transfers → Optimizer → Register Transfers

Outline:
- What is a Compiler?
- The Birth of a Compiler
- The Structure of Modern Compilers
- Modern Challenges
- Conclusions
Uday Khedker
IIT Bombay

Talk Title: Compiling-Challenges

Topic:
Outline

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions

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
The Birth of a Compiler

Outline

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions

Compilation Models

Aho Ullman Model

Front End

AST

Optimizer

Target Indep. IR

Code Generator

Target Program

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

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

Davidson Fraser Model

Front End

AST

Expander

Register Transfers

Optimizer

Register Transfers

Recognizer

Target Program
Typical Front Ends

Parser
## Typical Front Ends

- **Source Program**
- **Scanner**
- **Parser**
- **Tokens**

**Outline**
- What is a Compiler?
- The Birth of a Compiler
- The Structure of Modern Compilers
- Modern Challenges
- Conclusions
Typical Front Ends

- Source Program
- Scanner
- Tokens
- Parse Tree
- AST
- Semantic Analyzer
- Parser
- AST or Linear IR
- Symbol Table
What is a Compiler?

The Birth of a Compiler

The Structure of Modern Compilers

Modern Challenges

Conclusions

Diagram:

- Source Program
- Parser
- Tokens
- Parse Tree
- AST
- AST or Linear IR + Symbol Table
- Semantic Analyzer
- Symtab Handler
- Error Handler

Flow:

1. Source Program -> Scanner
2. Scanner -> Tokens
3. Tokens -> Parser
4. Parser -> Parse Tree
5. Parse Tree + Symtab Handler -> AST
6. AST -> Error Handler
7. AST or Linear IR + Symbol Table
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

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

Uday Khedker
IIT Bombay

Talk Title:
Compiling-Challenges

Topic:
Outline

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
The GNU Tool Chain for C

Source Program

gcc

Target Program
The GNU Tool Chain for C

Source Program

gcc

Target Program

cc1
The GNU Tool Chain for C

Source Program

```
gcc
```

Target Program

```
cc1
```

```
cpp
```
The GNU Tool Chain for C

Source Program

Target Program

gcc

cc1

as

cpp
The GNU Tool Chain for C

Source Program

<table>
<thead>
<tr>
<th>gcc</th>
</tr>
</thead>
</table>

Target Program

<table>
<thead>
<tr>
<th>cc1</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpp</td>
</tr>
<tr>
<td>as</td>
</tr>
<tr>
<td>ld</td>
</tr>
</tbody>
</table>
The GNU Tool Chain for C

Source Program

gcc

cc1

cpp

as

ld

glibc/newlib

Target Program
The GNU Tool Chain for C

Source Program

- cc1
- cpp
- as
- ld
- glibc/newlib

Target Program

gcc
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

Compiler Generation Framework:
- Parser
- Gimplifier
- Tree SSA Optimizer
- Expander
- Optimizer
- Recognizer

Source Program ➔ Generated Compiler (cc1) ➔ Assembly Program
The Architecture of GCC

- **Input Language**
- **Compiler Generation Framework**
  - Language Specific Code
  - Language and Machine Independent Generic Code
  - Machine Dependent Generator Code
  - Machine Descriptions

- **Selected**
- **Copied**
- **Generated**

- **Parser**
- **Gimplifier**
- **Tree SSA Optimizer**
- **Expander**
- **Optimizer**
- **Recognizer**

- **Source Program**
- **Generated Compiler (cc1)**
- **Assembly Program**

- **Target Name**
The Architecture of GCC

Input Language

Compiler Generation Framework

Language Specific Code

Language and Machine Independent Generic Code

Machine Dependent Generator Code

Machine Descriptions

Selected

Copied

Copied

Generated

Generated

Development Time

Build Time

Use Time

Parser

Gimplifier

Tree SSA Optimizer

Expander

Optimizer

Recognizer

Source Program

Generated Compiler (cc1)

Assembly Program

Target Name
GCC Retargetability Mechanism

- Input Language
- Compiler Generation Framework
- Target Name
- Development Time
- Build Time
- Use Time

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

- Selected
- Copied
- Copied
- Generated
- Generated

- Parser
- Gimplifier
- Tree SSA Optimizer
- Expander
- Optimizer
- Recognizer

Generated Compiler

Uday Khedker
IIT Bombay

Talk Title:
Compiling-Challenges

Topic:
Outline
What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions

55/83
GCC Retargetability Mechanism

- Input Language
- Compiler Generation Framework
- Target Name


Parser → Gimplifier → Tree SSA Optimizer → Expander → Optimizer → Recognizer → Generated Compiler

- Development Time
- Build Time
- Use Time

Gimple → IR-RTL
IR-RTL → ASM

Outline:
1. What is a Compiler?
2. The Birth of a Compiler
3. The Structure of Modern Compilers
4. Modern Challenges
5. Conclusions
**Talk Title:** Compiling-Challenges

**Outline**

- What is a Compiler?
- The Birth of a Compiler
- The Structure of Modern Compilers
- Modern Challenges
- Conclusions

---

**GCC Retargetability Mechanism**

- Input Language
- Compiler Generation Framework
- Target Name

**Language Specific Code**
- Selected

**Language and Machine Independent Generic Code**
- Copied

**Machine Dependent Generator Code**
- Copied

**Machine Descriptions**
- Generated

**Compiler Generation Framework**

- **Parser**
- **Gimplifier**
- **Tree SSA Optimizer**
- **Expander**
- **Optimizer**
- **Recognizer**

**Generated Compiler**

- **Gimple → PN**
- **PN → IR-RTL**
- **IR-RTL → ASM**

**Development Time**

- **Gimple → IR-RTL**
- **IR-RTL → ASM**

**Build Time**

- **Use Time**
GCC Retargetability Mechanism

1. **Input Language** → **Language Specific Code**
   - Selected
   - Copied

2. **Language and Machine Independent Generic Code**
   - Copied

3. **Machine Dependent Generator Code**
   - Generated

4. **Machine Descriptions**
   - Generated

5. **Target Name** → **Compiler Generation Framework**
   - Development Time
   - Build Time
   - Use Time

6. **Generated Compiler**

7. **Gimple → PN**
   - PN → IR-RTL
   - IR-RTL → ASM

8. **Gimple → IR-RTL**
   - IR-RTL → ASM
GCC Retargetability Mechanism

Input Language → Compiler Generation Framework → Target Name

Language Specific Code → Language and Machine Independent Generic Code → Machine Description

Select → Copy → Copy → Generated

Gimple → PN + PN → IR-RTL + IR-RTL → ASM

Gimple → IR-RTL + IR-RTL → ASM
The generated compiler uses an adaptation of the Davidson Fraser model

- Generic expander and recognizer
- Machine specific information is isolated in data structures
- Generating a compiler involves generating these data structures
The LLVM Tool Chain for C

Source Program

clang

Target Program

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
The LLVM Tool Chain for C

Source Program

clang

Target Program

clag -cc1
The LLVM Tool Chain for C

Uday Khedker
IIT Bombay

Talk Title:
Compiling-Challenges

Topic:
Outline

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions

Source Program

clang -cc1

clang -E

Target Program
The LLVM Tool Chain for C

Source Program

clang

Target Program

clang -cc1

clang -E

llvm-as
The LLVM Tool Chain for C

Source Program

clang

clang -cc1

llvm-as

lld

clang -E

Target Program
The LLVM Tool Chain for C

Source Program

clang

clang -cc1

clang -E

llvm-as

glibc/newlib

lld

Target Program
The LLVM Tool Chain for C

Source Program

clang

clang -cc1

llvm-as

lld

glibc/newlib

clang -E

Target Program

Outline

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
Simplified x86 Target Definition

Reproduced from https://www.aosabook.org/en/llvm.html
Building a Compiler: Terminology

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

The built compiler itself runs on HS and generates executables that run on TS.
### Variants of Compiler Builds

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

**Example**

**Native i386**: built on i386, hosted on i386, produces i386 code.

**Sparc cross on i386**: built on i386, hosted on i386, produces Sparc code.
T Notation for a Compiler
T Notation for a Compiler

input language

C

i386

cc

i386

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
T Notation for a Compiler
T Notation for a Compiler

input language

output language

implementation or execution language

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
Talk Title: Compiling-Challenges

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions

T Notation for a Compiler

- input language
- output language
- implementation or execution language
- name of the translator
- "C"
- "i386"
- "cc"
- "i386"
Bootstrapping: The Conventional View

- Assembly language
- Machine language
- m/c
Bootstrapping: The Conventional View
Bootstrapping: The Conventional View

input language
implementation language
output language

Level 0 C

C₀

m/c

ass
Bootstrapping: The Conventional View

input language

implementation language

output language

C₀

ass

m/c

m/c

m/c

Level 0 C
Bootstrapping: The Conventional View

- What is a Compiler?
- The Birth of a Compiler
- The Structure of Modern Compilers
- Modern Challenges
- Conclusions
Bootstrapping: The Conventional View

Level 1 C

C₀

ass

C₁

input language

implementation language

output language

m/c

m/c
Bootstrapping: The Conventional View

Level n C

input language

C_n

C_{n-1}

implementation language

output language

m/c

input language
Bootstrapping: The Conventional View

Level n C

input language

C_{n-1}

C_n

implementation language

m/c

C_{n-2}

output language

input language

m/c

output language
• Language need not change, but the compiler may change
  Compiler is improved, bugs are fixed and newer versions are released
• To build a new version of a compiler given a built old version:
  o Stage 1: Build the new compiler using the old compiler
  o Stage 2: Build another new compiler using compiler from stage 1
  o Stage 3: Build another new compiler using compiler from stage 2
  Stage 2 and stage 3 builds must result in identical compilers
⇒ Building cross compilers stops after Stage 1!
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
  o The design and implementation of a compiler is a great application of software engineering
  o Makes practical application of deep theory and algorithms and rich data structures
  o Uses rich features of computer architecture
Translating Deep Theory into Affordable Practice

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

Credits: Adapted from the slides of Prof. Y. N. Srikant, IISc Bangalore
• 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.
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
  o “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
  o “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
  o 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
  o “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
  o GPUs, Many core processors, Embedded processors

• Problem sizes have changed significantly
  o Programs running in millions of lines of code

• Expectations have changed significantly
  o Interprocedural analysis and optimization, validation, reverse engineering, parallelization

• Analysis techniques have changed significantly
  o Parsing, Data flow analysis, Parallism Discovery, Heap Analysis
Compilation for Embedded Processors

Application

- Standard
- Non-Standard

Architecture

- Standard
- Non-Standard

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
Compilation for Embedded Processors

Application

Language

Compiling Techniques

Architecture

Standard
Non-Standard
Standard
Non-Standard
Standard
Non-Standard
Standard
Non-Standard

Uday Khedker
IIT Bombay

Talk Title:
Compiling-Challenges

Topic:
Outline
What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions
Compilation for Embedded Processors

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

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

- MACs, Special loop instructions
• Setting arithmetic modes, circular addressing, special loop instructions
Modern Challenges: Design issues

• The IR interface
  What to export? What to hide?
  The most challenging component to design and implement in a compiler is the IR handler

• Retargetability
  Extending to the new version of a processor?
  Extending to a new processor?
Modern Challenges: Improving Performance of Programs

• Scaling analysis to large programs without losing precision
  o Interprocedural analysis
  o Pointer analysis
Modern Challenges: Improving Performance of Programs

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

- Increasing the precision of analysis
  - How to interleave difference analysis to benefit from each other?
  - How to exclude infeasible interprocedural paths?
Modern Challenges: Improving Performance of Programs

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

- Increasing the precision of analysis
  - How to interleave difference analysis to benefit from each other?
  - How to exclude infeasible interprocedural paths?

- Combining static and dynamic analysis
  - Using statically computed information for optimization at run time
  - Using run time information for improving optimizations in the next compilation
    (Profile guided optimization aka feedback driven optimization)
Modern Challenges: Improving Performance of Programs

• Scaling analysis to large programs without losing precision
  ○ Interprocedural analysis
  ○ Pointer analysis

• Increasing the precision of analysis
  ○ How to interleave difference analysis to benefit from each other?
  ○ How to exclude infeasible interprocedural paths?

• Combining static and dynamic analysis
  ○ Using statically computed information for optimization at run time
  ○ Using run time information for improving optimizations in the next compilation
    (Profile guided optimization aka feedback driven optimization)

• Inventing more effective optimizations
Modern Challenges: Improving Performance of Programs

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

- Full Employment Guarantee Theorem for Compiler Writers
  (https://en.wikipedia.org/wiki/Full_employment_theorem)

- The notion of “best” compiler cannot exist and there is endless scope to keep improving
  ⇒ For every compiler, a better compiler can be written

- Inventing more effective optimizations
Modern Challenges: Language Issues

- How to efficiently compile
  - Dynamic features such as closures, higher order functions (eg. eval in Javascript)
  - Exceptions

- What guarantees to give in the presence of undefined behaviour
  - Memory accesses such as array access out of bound

- Designing analyses for features supporting parallelism
  - Doall, Async, Threads, Synchronization, Fork/Join, Lock/Unlock, Mutex, Semaphores
  - Some features enable parallelism in a sequential language whereas some enforce sequentiality on essentially parallel execution

- Designing analyses for extracting parallelism
Modern Challenges: Target Machine Issues

How to exploit

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

General strategy: Hardware software co-design
Modern Challenges: Target Machine Issues

The crux of the matter

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

• Software view is stable, hardware is disruptive
A concurrent program

Initially $X = Y = 0$

$\begin{align*}
a &= X \\
Y &= 1
\end{align*}$

$\begin{align*}
b &= Y \\
\text{if}(b) X &= 1
\end{align*}$

- Variables $a$ and $b$ are thread-local variables
- Variables $X$ and $Y$ are shared global variables
Architecture Feature Influencing Programming Language

A concurrent program

Initially \( X = Y = 0 \)

- \( a = X \)
- \( b = Y \)
- \( Y = 1 \)
- \( \text{if}(b) \ X = 1 \)

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

Sequential Consistency preserves program order

\[
\begin{align*}
& a = X \\
& Y = 1 \\
& b = Y \\
& b? \ X = 1 \\
\end{align*}
\]
A concurrent program

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

- **Strict consistency.** Result of writes are available instantaneously
  - Both prints of $a$ must read the value 1
  - $\Rightarrow$ Only one result is possible: 1, 1

- **Sequential consistency.** Result of writes may be available with a delay
  - The first print of $a$ may read 0
  - $\Rightarrow$ Both 0, 1 and 1,1 are possible
A concurrent program

Initially \( X = Y = 0 \)

- \( a = X \)
- \( b = Y \)
- \( Y = 1 \)
- \( \text{if}(b) \ X = 1 \)

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

Relaxed Memory Consistency allows violating program order

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

A concurrent program

Initially $X = Y = 0$

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

Relaxed Memory Consistency allows violating program order

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

Why is this useful?

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

- $Y = 1$
- $b = Y$
- $b \land X = 1$
- $a = X$
- $a = b = 1$
Modern Challenges: Providing Guarantees

- Correctness of optimizations
  - Hard even for machine independent optimizations
  - Verification of a production optimizing compiler is a pipe dream
    Requires proving the correctness of translation of ALL programs
  - Compiler validation is more realistic, and yet not achieved fully
    Allows proving the correctness of translation of A program

- Interference with Security
  - Optimizations disrupt memory view
    Correctness is defined in terms of useful states
    Clearing stack location by writing all zeros is dead code
  - Optimizations also disrupt timing estimates
Compiler Verification

Formalize and verify the following diagram for every source program $P$

\[ P \quad \xrightarrow{\text{comp}} \quad \text{comp}(P) \]

\[ s\text{Mean}(P) \quad \xleftarrow{\text{abstract}} \quad t\text{Mean(\text{comp}(P))} \]

$\text{comp}$ represents the transformation performed by

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

Is the model faithful?

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

Formalize and verify the following diagram for every source program P

\[
\begin{align*}
\text{P} & \quad \Rightarrow \quad \text{comp(P)} \\
\text{y} = 5 & \quad \Rightarrow \quad \text{sMean(P)} \quad \Rightarrow \quad \text{tMean(comp(P))} \\
\text{x} = y + 1 & \quad \Rightarrow \quad \text{abstract} \\
\text{y} = 5 & \quad \Rightarrow \quad \text{x} = 6 \\
\end{align*}
\]

\[
\begin{align*}
\text{li $t0, 5$} & \\
\text{sw $t0, 4($fp$)$} & \\
\text{lw $t0, 4($fp$)$} & \\
\text{addi $t0, 1$} & \\
\text{sw $t0, 8($fp$)$} & \\
\text{M[4($fp$)] = 5} & \\
\text{M[8($fp$)] = 6} & \\
\end{align*}
\]

\text{comp} represents the transformation performed by

- a compiler \textbf{(harder problem)}, or
- a model of the compiler \textbf{(easier)}

Is the model faithful?

Credits: Adapted from the slides of Prof. Amitabha Sanyal, IIT Bombay
Difficulties in Compiler Verification

• Complexity
  ○ Requires reasoning about actual compiler implementation.
  ○ Requires reasoning about the behaviour of the compiler for an infinite number of programs and their translations.

• Automation - unlikely

• Proof reuse?

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

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

\[ P \xrightarrow{\text{comp}} \text{comp}(P) \]

\[ \text{sMean}(P) \xleftarrow{\text{abstract}} \text{tMean}(\text{comp}(P)) \]

\( \text{comp} \) represents the transformation performed by:

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

Is the model faithful?

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

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

$$y = 5$$
$$x = y + 1$$

$P$ → $\text{comp} (P)$ → $\text{sMean} (P) → \text{tMean} (\text{comp} (P))$

$y = 5$
$x = 6$

$\text{li} \ t0, 5$
$\text{sw} \ t0, 4(\text{fp})$
$\text{lw} \ t0, 4(\text{fp})$
$\text{addi} \ t0, 1$
$\text{sw} \ t0, 8(\text{fp})$

$M[4(\text{fp})] = 5$
$M[8(\text{fp})] = 6$

$\text{comp}$ represents the transformation performed by

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

Is the model faithful?

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

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

• Automation - likely.

Credits: Adapted from the slides of Prof. Amitabha Sanyal, IIT Bombay
Modern Challenges: New Expectations

- New application domains bringing new challenges

- What are the underlying abstractions of the domains that should become first class citizens in a programming language?
  - Language design and compilers for machine learning algorithms?
  - Language design and compilers for streaming applications?

- Can machine learning algorithms help compilers create new optimizations?
  - Can human ingenuity in design of novel algorithms be replaced by machine learning?
    Need explanability for guaranteeing soundess of new optimizations
    Known cost based optimizations have a better chance with machine learning
  - Can compilers learn from the programs they have compiled and become “better” over time?
Conclusions
The Wonder Element of FORTRAN

- Expressiveness Vs. Efficiency conflict
  - Efficiency of programming and reach of programming, OR
  - Efficiency of program execution and resource utilization

- FORTRAN: The triumph of the genius of AND over the tyranny of OR
The Wonder Element of FORTRAN

• Expressiveness Vs. Efficiency conflict
  ○ Efficiency of programming and reach of programming, OR
  ○ Efficiency of program execution and resource utilization

• FORTRAN: The triumph of the genius of AND over the tyranny of OR

• The software equivalent of a transistor
The Challenge Ahead

- Expressiveness Vs. Efficiency conflict due to the problem of scale
The Challenge Ahead

- Expressiveness Vs. Efficiency conflict due to the problem of scale
- Have we reached the Von Neumann bottleneck?
The Challenge Ahead

- Expressiveness Vs. Efficiency conflict due to the problem of scale

- Have we reached the Von Neumann bottleneck?
  Backus argued so over three decades ago!
The Challenge Ahead

• Expressiveness Vs. Efficiency conflict due to the problem of scale

• Have we reached the Von Neumann bottleneck?
  Backus argued so over three decades ago!

• At an abstract level, the status of compilers is similar to those in the John Backus era
  ○ Architectures not understood well enough for exploitation by compilers
  ○ Architectures influencing language features
  ○ Comparison with assembly
  ○ No past success story
Uday Khedker
IIT Bombay

Talk Title: Compiling-Challenges

Topic:
Outline

What is a Compiler?
The Birth of a Compiler
The Structure of Modern Compilers
Modern Challenges
Conclusions

Interesting Reads Available Online

- The Computer History Museum (www.computerhistory.org)
  - FORTRAN examples by John Backus
  - Array copy example by Frances Allen
  - FORTRAN expression handling explanation by David Padua

- Proceedings of the History of Programming Languages conferences: https://dl.acm.org/conference/hopl/proceedings
The Moral of the Story

Achieving Performance

Expressiveness (Rich abstractions)
Generality (Retargetability, upgrades and enhancements)
Providing Guarantees (Correctness, robustness, security)
The Moral of the Story

Achieving Performance

The Range of Modern compilers

Expressiveness (Rich abstractions)
Generality (Retargetability, upgrades and enhancements)
Providing Guarantees (Correctness, robustness, security)
Achieving Performance

Expressiveness (Rich abstractions)
Generality (Retargetability, upgrades and enhancements)
Providing Guarantees (Correctness, robustness, security)

The Moral of the Story

Getting here is the real challenge!
The Moral of the Story

The world awaits another John Backus to give us the next breakthrough!

Achieving Performance

Expressiveness (Rich abstractions)
Generality (Retargetability, upgrades and enhancements)
Providing Guarantees (Correctness, robustness, security)
Thank You!
Last But Not the Least

Thank You!

Contacting me :

• uday@cse.iitb.ac.in

• http://www.cse.iitb.ac.in/~uday