Introduction to Compilers

Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu
Why are we here?
Why Study Compilers?

It’s cool to ...

• build a large, complicated, ambitious software system, consisting of thousands of lines of code, over an entire semester.
• use all the good stuff you learned in Algorithms.
• learn how programming languages work.
• see computer science theory come to life.
• learn how to implement a programming language.

Building a compiler will make you a better programmer. I guarantee it.

If you finish this project, I will be very proud of you.

More importantly: if you finish this project, you will be VERY proud of yourself.
A long time ago….

All you could do was push buttons and flip switches.
A long time ago....

Code and data was (and still is) just an electrical signal.
A long time ago....

We can think of changes in current as binary digits.

\[ 1010 \ 1001 \ 0000 \ 0011 \]
\[ 1000 \ 1101 \ 0100 \ 0001 \ 0000 \ 0000 \]
A long time ago....

We can think of changes in current as binary or hexadecimal digits.

1010 1001 0000 0011
1000 1101 0100 0001 0000 0000

A9 03
8D 41 00

But how could you store it?
A long time ago....

The first storage

(Taken from Chris Algozzine’s office.)

(Written by Bill Gates and Paul Allen)

Now the pieces are in place . . .
A long time ago....

FORTRAN
- FORMula TRANslator
- for Scientific computing
- by John Backus from IBM

COBOL
- COMMON Business Oriented Language
- for business computing
- by Rear Admiral Grace Hopper
  - Coins the term “Compiler” and writes the first one
  - Math faculty at Vassar College

Jumping ahead a few decades...
Compiler vs. Interpreter

Compiler
- a **program** that...
  - translates one language to another that...
  - outputs an executable program...
  - that is expected to be *better* (somehow) than the input.
  - Examples include C, C++, Java, TypeScript

Interpreter
- a **program** that...
  - reads an executable program in some language...
  - and runs that program.
  - Examples include JavaScript, BASIC, Python
Before and After Compilation

**Before:** Three higher-level languages

```
10 A = 3
20 X = 1
30 print X
40 X = X + 1
50 IF X <> A THEN GOTO 30
60 PRINT “DONE”
```

```
    int limit = 3;
    int val = 1;
    repeat
        {          
            console.write(val);
            ++val;
        } until (val == limit)
    console.write(“DONE”);
```

```
var limit = 3;
var val = 1;
repeat
    {         
        alert(val);
        ++val;
    } while (val <> limit);
    alert(“DONE”);
```

What languages are these?
### Before and After Compilation

**During:** Intermediate Representation (6502a Assembly. Ms-IL and Java bytecodes are similar.)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Acc</th>
<th>Mem[41]</th>
<th>Mem[40]</th>
<th>Mem[40]++</th>
<th>X</th>
<th>loop</th>
<th>Y</th>
<th>X</th>
<th>Z</th>
<th>if Z == 0 goto loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>lda #$3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sta $0041</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lda #$1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sta $0040</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop ldy $0040</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldx #$01</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inc $0040</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldx $0040</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cp px $0041</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bne loop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lda #$44</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sta $0042</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lda #$4F</td>
<td>4F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sta $0043</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lda #$4E</td>
<td>4E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sta $0044</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lda #$45</td>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sta $0045</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lda #$00</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sta $0046</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldx #$02</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ldy #$42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>brk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|        | 0000 | LDA #$03 | A9  03 |
|        | 0002 | STA $0041 | 8D 41 00 |
|        | 0005 | LDA #$01 | A9  01 |
|        | 0007 | STA $0040 | 8D 40 00 |
|        | 000A | LOOP     | LO | 40 00 |
|        | 000D | LDX #$01 | A2  01 |
|        | 000F | SYS      | FF |
|        | 0010 | INC $0040 | EE 40 00 |
|        | 0013 | LDX $0040 | AE 40 00 |
|        | 0016 | CPX $0041 | EC 41 00 |
|        | 0019 | BNE LOOP  | D0  EF |
|        | 001B | LDA #$44 | A9  44 |
|        | 001D | STA $0042 | 8D 42 00 |
|        | 0020 | LDA #$4F | A9  4F |
|        | 0022 | STA $0043 | 8D 43 00 |
|        | 0025 | LDA #$4E | A9  4E |
|        | 0027 | STA $0044 | 8D 44 00 |
|        | 002A | LDA #$45 | A9  45 |
|        | 002C | STA $0045 | 8D 45 00 |
|        | 002F | LDA #$00 | A9  00 |
|        | 0031 | STA $0046 | 8D 46 00 |
|        | 0034 | LDX #$02 | A2  02 |
|        | 0036 | LDY #$42 | A0  42 |
|        | 0038 | SYS      | FF |
|        | 0039 | BRK      | 00 |
# Before and After Compilation

**During:** Intermediate Representation (6502a Assembly. Ms-IL and Java bytecodes are similar.)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>Assembler Code</th>
<th>Machine Code</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>lda #$3</td>
<td>Acc = 3</td>
<td>0000</td>
<td>A9 03</td>
<td></td>
</tr>
<tr>
<td>sta $0041</td>
<td>Mem[41] = 3</td>
<td>0002</td>
<td>8D 41 00</td>
<td></td>
</tr>
<tr>
<td>lda #$1</td>
<td>Acc = 1</td>
<td>1010</td>
<td>1001 0000 0011</td>
<td></td>
</tr>
<tr>
<td>sta $0040</td>
<td>Mem[40] = 1</td>
<td>1000 1101 0100 0001 0000 0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop ldy $0040</td>
<td>Y = Mem[40]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>X = 1</td>
<td>000F</td>
<td>SYS  FF</td>
<td></td>
</tr>
<tr>
<td></td>
<td>System Call</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inc $0040</td>
<td>Mem[40]++</td>
<td>0010</td>
<td>INC $0040 EE 40 00</td>
<td></td>
</tr>
<tr>
<td>ldx $0040</td>
<td>X = Mem[40]</td>
<td>0013</td>
<td>LDX $0040 AE 40 00</td>
<td></td>
</tr>
<tr>
<td>cpx $0041</td>
<td>Z bit = (x == Mem[41]) if z == 0 goto loop</td>
<td>0016</td>
<td>CPX $0041 EC 41 00</td>
<td></td>
</tr>
<tr>
<td>bne loop</td>
<td></td>
<td>0019</td>
<td>BNE LOOP D0 EF</td>
<td></td>
</tr>
<tr>
<td>lda #$44</td>
<td>Acc = $44 (“D”)</td>
<td>001B</td>
<td>LDA #$44 A9 44</td>
<td></td>
</tr>
<tr>
<td>sta $0042</td>
<td>Mem[42] = $44</td>
<td>001D</td>
<td>STA $0042 8D 42 00</td>
<td></td>
</tr>
<tr>
<td>lda #$4F</td>
<td>Acc = $4F (“O”)</td>
<td>0020</td>
<td>LDA #$4F A9 4F</td>
<td></td>
</tr>
<tr>
<td>sta $0043</td>
<td>Mem[43] = $4F</td>
<td>0022</td>
<td>STA $0043 8D 43 00</td>
<td></td>
</tr>
<tr>
<td>lda #$4E</td>
<td>Acc = $4E (“N”)</td>
<td>0025</td>
<td>LDA #$4E A9 4E</td>
<td></td>
</tr>
<tr>
<td>sta $0044</td>
<td>Mem[44] = $4E</td>
<td>0027</td>
<td>STA $0044 8D 44 00</td>
<td></td>
</tr>
<tr>
<td>lda #$45</td>
<td>Acc = $45 (“E”)</td>
<td>002A</td>
<td>LDA #$45 A9 45</td>
<td></td>
</tr>
<tr>
<td>sta $0045</td>
<td>Mem[45] = $45</td>
<td>002C</td>
<td>STA $0045 8D 45 00</td>
<td></td>
</tr>
<tr>
<td>lda #$00</td>
<td>Acc = $00 (null)</td>
<td>002F</td>
<td>LDA #$00 A9 00</td>
<td></td>
</tr>
<tr>
<td>sta $0046</td>
<td>Mem[46] = $00</td>
<td>0031</td>
<td>STA $0046 8D 46 00</td>
<td></td>
</tr>
<tr>
<td>ldx #$02</td>
<td>X = 2</td>
<td>0034</td>
<td>LDX #$02 A2 02</td>
<td></td>
</tr>
<tr>
<td>ldy #$42</td>
<td>Y = $42 (address)</td>
<td>0036</td>
<td>LDY #$42 A0 42</td>
<td></td>
</tr>
<tr>
<td>sys</td>
<td>System call</td>
<td>0038</td>
<td>SYS FF</td>
<td></td>
</tr>
<tr>
<td>brk</td>
<td>Break</td>
<td>0039</td>
<td>BRK 00</td>
<td></td>
</tr>
</tbody>
</table>
Before and After Compilation

10 A = 3
20 X = 1
30 print X
40 X = X + 1
50 IF X <> A THEN GOTO 30
60 PRINT "DONE"

lda #$3  
sta $0041

lda #$1  
sta $0040

loop ldy $0040 
  ldx #$01
  sys

inc $0040
  ldx $0040

cmpeq $0041
  bne loop

lda #$44
sta $0042
lda #$4F
sta $0043
lda #$4E
sta $0044
lda #$45
sta $0045
lda #$00
sta $0046
lda #$02
ldi #$42
sys

brk
Aside: von Neumann Architecture

**lda #\$3**

**sta $0041**

$1000 1101 0100 0001 0000 0000$

$0000 0011$

**loc 0041:**

$0000 0011$

**STA op code (8D)**

**Arithmetic and logic unit**

**Control unit**

**Accumulator (register):** $0000 0011$

**Results of operations**

**Instructions and data** across the front-side bus

**Memory (stores both instructions and data)**

**Input and output devices**

$10 \times A = 3$

15
### Before and After Compilation

#### Listing 1

```
10 A = 3
20 X = 1
30 print X
40 X = X + 1
50 IF X <> A THEN GOTO 30
60 PRINT "DONE"
```

#### Assembly Code 1

```
lda #$3
sta $0041
```

#### Machine Code 1

```
A9 03 1010 1001 0000 0011
8D 41 00 1000 1101 0100 0001 0000 0000
```

#### Listing 2

```
lda #$1
sta $0040
```

#### Assembly Code 2

```
A9 01 1010 1001 0000 0001
8D 40 00 1000 1101 0100 0000 0000 0000
```

#### Machine Code 2

```
A9 01 1010 1001 0000 0001
8D 40 00 1000 1101 0100 0000 0000 0000
```

#### Listing 3

```
loop ldy $0040
    ldx #$01
    sys
```

#### Assembly Code 3

```
AC 40 00 1010 1100 0100 0000 0000 0000
A2 01 1010 0010 0000 0001
FF
```

#### Machine Code 3

```
AC 40 00 1010 1100 0100 0000 0000 0000
A2 01 1010 0010 0000 0001
FF
```

#### Listing 4

```
inc $0040
lda $0040
```

#### Assembly Code 4

```
EE 40 00 1010 1100 0100 0000 0000 0000
AE 40 00 1010 0010 0000 0001
```

#### Machine Code 4

```
EE 40 00 1010 1100 0100 0000 0000 0000
AE 40 00 1010 0010 0000 0001
```

#### Listing 5

```
cpx $0041
bne loop
```

#### Assembly Code 5

```
EC 41 00 1010 1100 0100 0000 0000 0000
D0 EF
```

#### Machine Code 5

```
EC 41 00 1010 1100 0100 0000 0000 0000
D0 EF
```

#### Listing 6

```
lda #$44
sta $0042
lda #$4F
sta $0043
lda #$4E
sta $0044
lda #$45
sta $0045
lda #$00
sta $0046
lda #$02
ldx #$42
sys
```

#### Assembly Code 6

```
A9 44
8D 42 00
A9 4F
8D 43 00
A9 4E
8D 44 00
A9 45
8D 45 00
A9 00
8D 46 00
A2 02
A0 42
```

#### Machine Code 6

```
A9 44
8D 42 00 1010 1101 0100 0000 0000 0000
A9 4F
8D 43 00 1010 1101 0100 0000 0000 0000
A9 4E
8D 44 00 1010 1101 0100 0000 0000 0000
A9 45
8D 45 00 1010 1101 0100 0000 0000 0000
A9 00
8D 46 00 1010 1101 0100 0000 0000 0000
A2 02
A0 42
```

#### Listing 7

```
brk
```

#### Assembly Code 7

```
00
```

#### Machine Code 7

```
00
```
Before and After Compilation

**After**: Machine Language

<table>
<thead>
<tr>
<th>A9 03 8D 41 00</th>
<th>A9 01 8D 40 00</th>
<th>AC 40 00</th>
<th>A2 01 FF EE 40 00</th>
<th>AE 40 00</th>
<th>EC 41 00</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF A9 44 8D 42 00</td>
<td>A9 4F 8D 43 00</td>
<td>A9 4E 8D 44 00</td>
<td>A9 45 8D 45 00</td>
<td>A9 00 8D 46 00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A2 02 A0 42 FF 00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How are we going to execute this?
## Before and After Compilation

### Source Code

**Before:** Three higher-level languages

```plaintext
10 A = 3
20 X = 1
30 print X
40 X = X + 1
50 IF X <> A THEN GOTO 30
60 PRINT “DONE”
```

```
int limit = 3;
int val = 1;
repeat
    { console.write(val);
      ++val;
    } until (val == limit)
    console.write(“DONE”);
```

```
var limit = 3;
var val = 1;
do
    { alert(val);
      ++val;
    } while (val <= limit);
alert(“DONE”);
```

**During:** Intermediate Representation (6502a Assembly. Ms-IL and Java bytecodes are similar.)

```plaintext
lda #3
sta $0041
lda #1
sta $0040
loop
ldy $0040
ldx #01
sys
inc $0040
ldx $0040
cpx $0041
bne loop
```

```
Acc = 3
Mem[41] = 3
0000 LDA #03 A9 03
0002 STA $0041 8D 41 00
```

```
Acc = 1
Mem[40] = 1
0005 LDA #01 A9 01
0007 STA $0040 8D 40 00
```

```
Y = Mem[40]
X = 1
System Call
Mem[40]++
X = Mem[40]
```

```
000A LOOP
000D LDY $0040 AC 40 00
000F SYS
```

```
0010 INC $0040 EE 40 00
0013 LDX $0040 AE 40 00
```

```
Z bit = (x = Mem[41])
if z == 0 goto loop
```

```
0016 CPX $0041 EC 41 00
0019 BNE LOOP D0 EF
```

```
Acc = $44 (“D”)
Mem[42] = $44
001B LDA $44 A9 44
001D STA $0042 8D 42 00
```

```
Acc = $4F (“O”)
Mem[43] = $4F
0020 LDA $4F A9 4F
0022 STA $0043 8D 43 00
```

```
Acc = $4E (“N”)
Mem[44] = $4E
0025 LDA $4E A9 4E
0027 STA $0044 8D 44 00
```

```
Acc = $45 (“E”)
Mem[45] = $45
002A LDA $45 A9 45
002C STA $0045 8D 45 00
```

```
Acc = $00 (null)
Mem[46] = $00
002F LDA $00 A9 00
0031 STA $0046 8D 46 00
```

```
X = 2
```

```
0034 LDX $02 A2 02
0036 LDY $42 A0 42
```

```
sys
System Call
```

```
0038 SYS FF
```

```
brk
Break
0039 BRK 00
```

**After:** Machine Language

```
A9 03 8D 41 00 A9 01 8D 40 00 AC 40 00 A2 01 FF EE 40 00 AE 40 00 EC 41 00 D0 EF A9 44 8D 42 00 A0 4F 8D 43 00 A9 4E 8D 44 00 A9 45 8D 45 00 A9 00 8D 46 00 A2 02 A0 42 FF 00
```
Super-High Level View

Source Code

Translator?

Machine Language
Super-High Level View

Source Code

Compiler

Machine Language

Goals
- recognize programs
- generate code
- manage storage
Super-High Level View

Source Code

Goals
- recognize **legal** programs
- generate **correct** code
- **efficiently** manage storage

Compiler

Errors and Warnings

Machine Language
Super-High Level View

Goals

- Front end translates code into IR
- Back end translates IR into Machine Language for the target platform.

Source Code

Front End

Back End

Intermediate Representation

Errors and Warnings

Machine Language
High Level View: Front End

- Source Code
  - Front End

Front End
- Recognize legal and illegal code
- Keep track of variables and types
- Report errors and warnings
- Produce the Intermediate Representation (IR)

- Consists of three parts:
High Level View: Front End

Source Code

Lex

Parse

Semantic Analysis
High Level View: Front End

Source Code

Lex

Parse

Semantic Analysis

Words

Sentences

Meaning
High Level View: Front End

Source Code

Lex

Parse

Semantic Analysis

Words

Sentences

Meaning

GLaDOS

cake
cake
ate
High Level View: Front End

Source Code

Lex

Parse

Semantic Analysis

Words

GLaDOS
cake
cake
ate

Sentences

ate cake GLaDOS

Meaning

GLaDOS ate cake
High Level View: Front End

Source Code

Lex

Parse

Semantic Analysis

Words

GLaDOS

cake

cake

ate

Sentences

GLaDOS ate cake

Meaning

GLaDOS

tate

cake

noun

verb

direct

object
Lexical Analysis

- Maps characters into an ordered stream of tokens
  
  \[
  x := x + y 
  \]
  
  becomes
  
  \[
  <id,x> <assign> <id,x> <add> <id,y> 
  \]

- **Typical tokens**: `id` `int` `while` `print` `if`

- Eliminates white space

- Report meaningful errors and warnings

- Produce a token stream

- Focus on words/lexemes/tokens
High Level View: Parse

Parse
- Recognize context-free syntax
- Recognize variables and type
- Report meaningful errors and warnings
- Produce a parse tree (aka: Concrete Syntax Tree)

- Focus on syntax
High Level View: Semantic Analysis

Semantic Analysis
- Produce a intermediate representation in the form of an Abstract Syntax Tree (AST)
- Check type
- Check scope
- Report meaningful errors and warning
- Focus on meaning
High Level View: Back End

Source Code → Lex → Token Stream → Parse → CST → Semantic Analysis → AST (IR)

Errors and Warnings

Symbol Table
High Level View: Back End

Code Generation
• Translates IR into Machine Language for the target platform
• Chose instructions for each IR operation
• Allocate registers
• Ensure conformance with target system

Source Code

Lex

Parse

Semantic Analysis

Errors and Warnings

Token Stream

CST

Symbol Table

AST (IR)

Code Generation

Machine Language
High Level View

Source Code → Lex → Parse → CST → Semantic Analysis

Token Stream → Parse

Errors and Warnings

Symbol Table

Project 1 → Project 2

Project 3 → AST (IR)

Project 4 → Code Generation → Machine Language
There’s more to the back end of a compiler:
- Lots of AST optimizations
- Instruction Selection
- Lots of instruction optimizations
- Register Allocation

Sadly, we don’t have time to implement these along with everything else in a single semester. But here’s a peek...
High Level View: Back End

Source Code

Lex

Token Stream

Parse

CST

Semantic Analysis

Errors and Warnings

Symbol Table

Instruction Optimizations

Instruction Selection

AST Optimizations

AST (IR)

Register Allocation

Code Generation

Machine Language
I wonder . . .

Source Code

Front End

How abstract should we get here?

Back End

Machine Language
I wonder . . .

Can we build $n \times m$ compilers with $n + m$ components?
Can we build $n \times m$ compilers with $n + m$ components? With the right Intermediate Representation, yes.
Example

```java
while (y < z) {
    int x = a + b;
    y += x;
}
```

<table>
<thead>
<tr>
<th>Lexical Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parsing</td>
</tr>
<tr>
<td>Semantic Analysis</td>
</tr>
<tr>
<td>IR Generation</td>
</tr>
<tr>
<td>IR Optimization</td>
</tr>
<tr>
<td>Code Generation</td>
</tr>
<tr>
<td>Optimization</td>
</tr>
</tbody>
</table>
Example

```c
while (y < z) {
    int x = a + b;
    y += x;
}
```

Lex: Did we get the **words** right?

This is based on an example from the great Alex Aiken.
Example

while (y < z) {
    int x = a + b;
    y += x;
}

Parse: Did we get the **sentences** right?
Example

```java
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

Assume these declarations so we can derive types.

SA: Did we get the **meaning** (scope and type) right?

This is based on an example from the great Alex Aiken.
Example

```c
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

Assume these declarations so we can derive types.

Lexical Analysis
Parsing
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

SA: Did we get the meaning (scope and type) right?

This is based on an example from the great Alex Aiken.
Example

```c
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

Assume these declarations so we can derive types.

SA: Did we get the **meaning** (scope and **type**) right?

This is based on an example from the great Alex Aiken.
Example

```c
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

Assume these declarations so we can derive types.

Lexical Analysis

Parsing

Semantic Analysis

IR Generation

IR Optimization

Code Generation

Optimization

SA: Did we get the **meaning** (scope and **type**) right?

This is based on an example from the great Alex Aiken.
Example

```c
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

ASS: Did we get the meaning (scope and type) right?

This is based on an example from the great Alex Aiken.
Example

```c
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

Assume these declarations so we can derive types.

Lexical Analysis

Parsing

Semantic Analysis

IR Generation

IR Optimization

Code Generation

Optimization

SA: Did we get the **meaning** (scope and **type**) right?

This is based on an example from the great Alex Aiken.
Example

```
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

} Assume these declarations so we can derive types.

Lexical Analysis
Parsing
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

SA: Did we get the **meaning** (scope and **type**) right?

This is based on an example from the great Alex Aiken.
Example

```c
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

Assume these declarations so we can derive types.

SA: Did we get the **meaning** (scope and **type**) right?

This is based on an example from the great Alex Aiken.
Example

```
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

Assume these declarations so we can derive types.

Lexical Analysis
Parsing
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization

SA: Did we get the **meaning** (scope and **type**) right?

This is based on an example from the great Alex Aiken.
Example

const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}

IR: Instead of using the AST as our IR we could use other code.

This is based on an example from the great Alex Aiken.
Example

```cpp
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

```plaintext
x = a + b
Loop: y = x + y
_tl = y < z
if _tl goto Loop
```

IR: A “hoisting” optimization

This is based on an example from the great Alex Aiken.
Example

```c
const int a = 3
const int b = 4
while (y < z) {
    int x = a + b;
    y += x;
}
```

Assume these declarations.

```
x  = a + b  7
```

Loop: `y  = x + y`

```
_tl = y < z
if _tl goto Loop
```

IR: A “constant folding” optimization

This is based on an example from the great Alex Aiken.
Example

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
sw $1, 0x7
Loop: add $4, $1, $4
slt $6, $1, $5
beq $6, loop
```

Code Gen: This is MIPS. We’ll use being 6502a op codes as our ML.

This is based on an example from the great Alex Aiken.
Example

```c
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
sw $1, 0x7
Loop: add $4, $1, $4
blt $1, $5, loop
```

ML Optimization: replace `set` and `branch` with `branch less than`.

This is based on an example from the great Alex Aiken.
Example

```
while (y < z) {
    int x = a + b;
    y += x;
}
```

// x = 7
A9 07 LDA #$07
8D 4E 00 STA $004E

LOOP:

// y = x + y
A9 07 LDA #$07
6D 4F 00 ADC $004F
8D 4F 00 STA $004F

// _t1 = y < x
AE 4F 00 LDX $004F
EC 50 00 CPX $0050

// if _t1 goto LOOP
D0 0F BNE $0026
00 BRK

Code Gen: 6502a op codes
The Goal

Source Code

Your Compiler

Machine Language