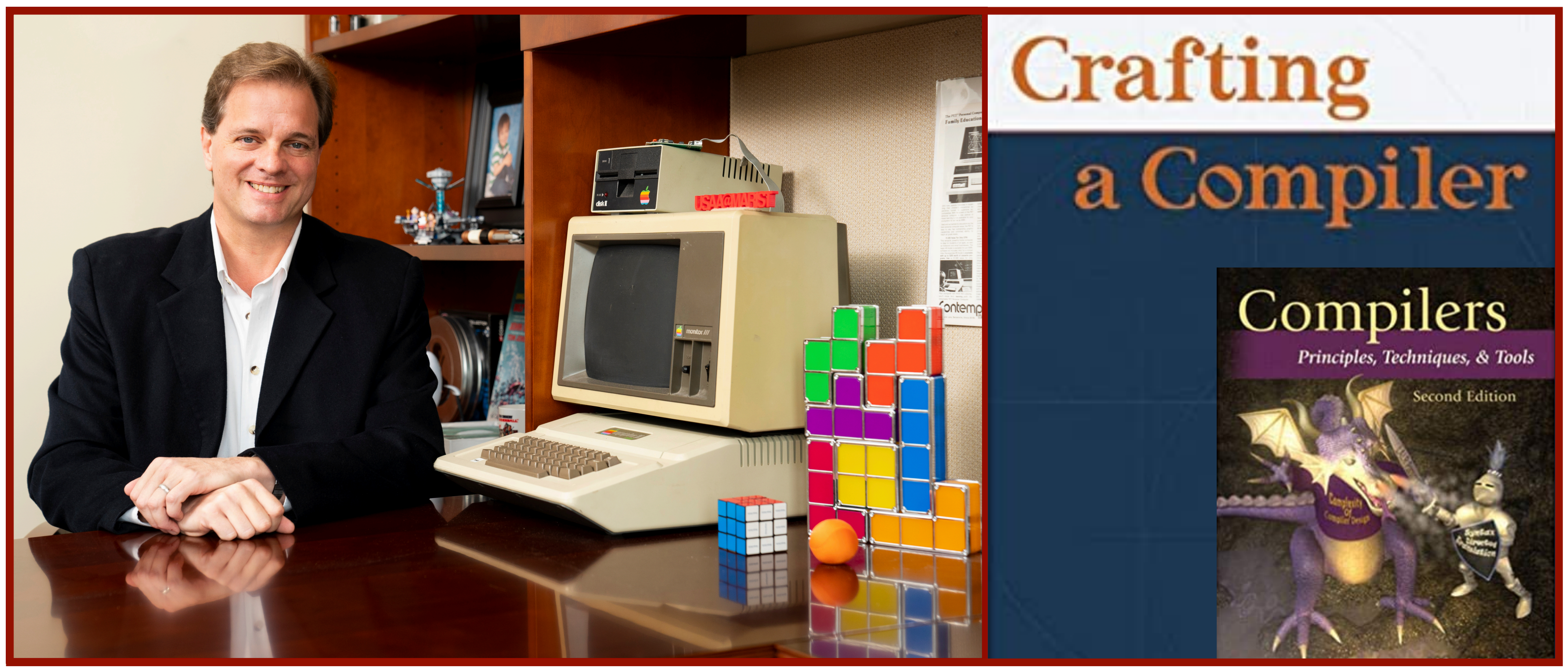
Introduction to Compilers



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.

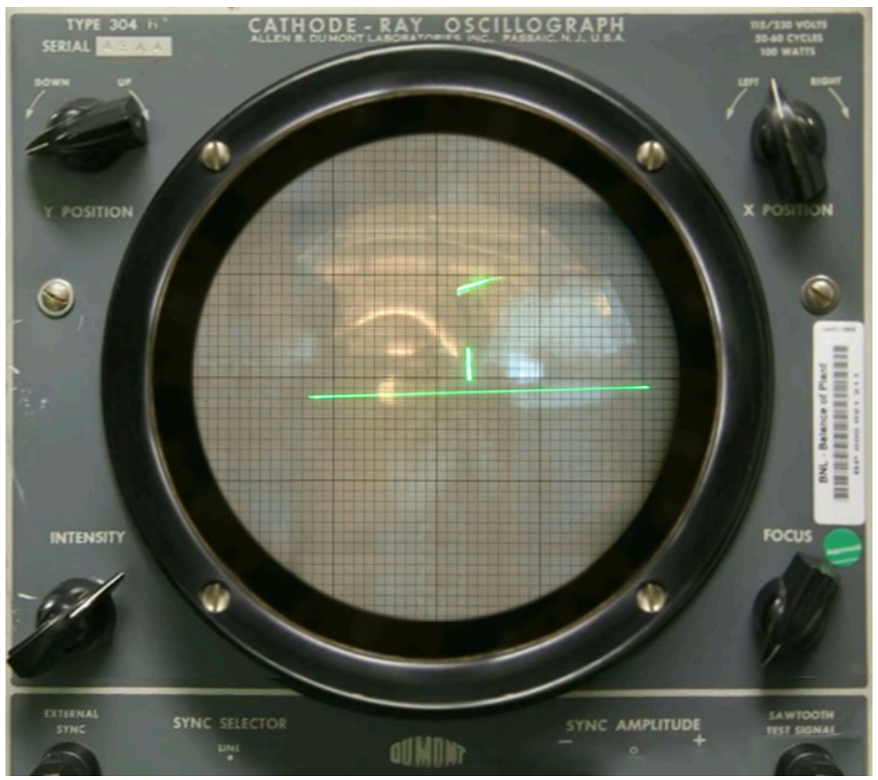
More importantly:
if you finish this project, you will be VERY proud of yourself.

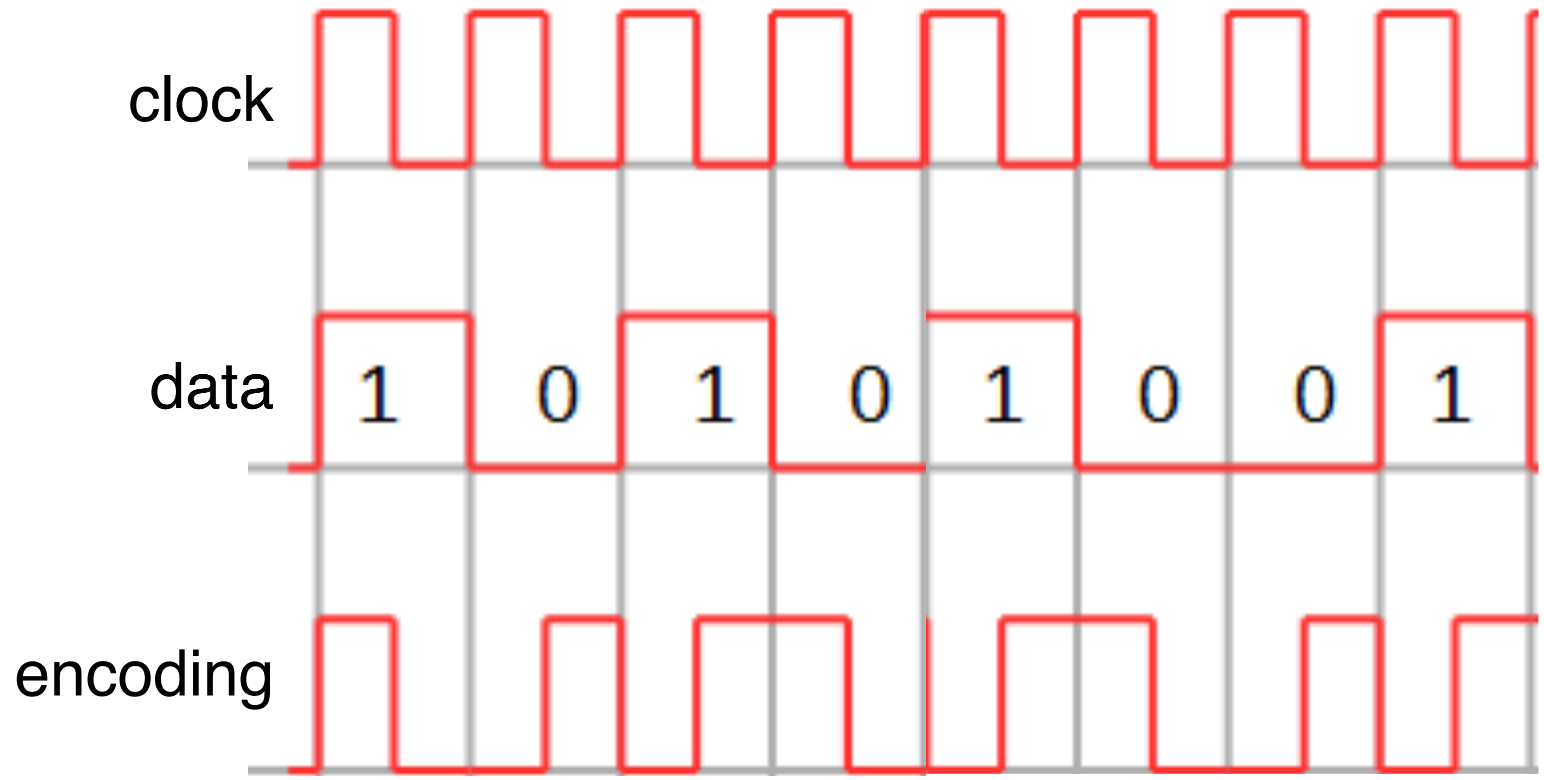
A long time ago….

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.

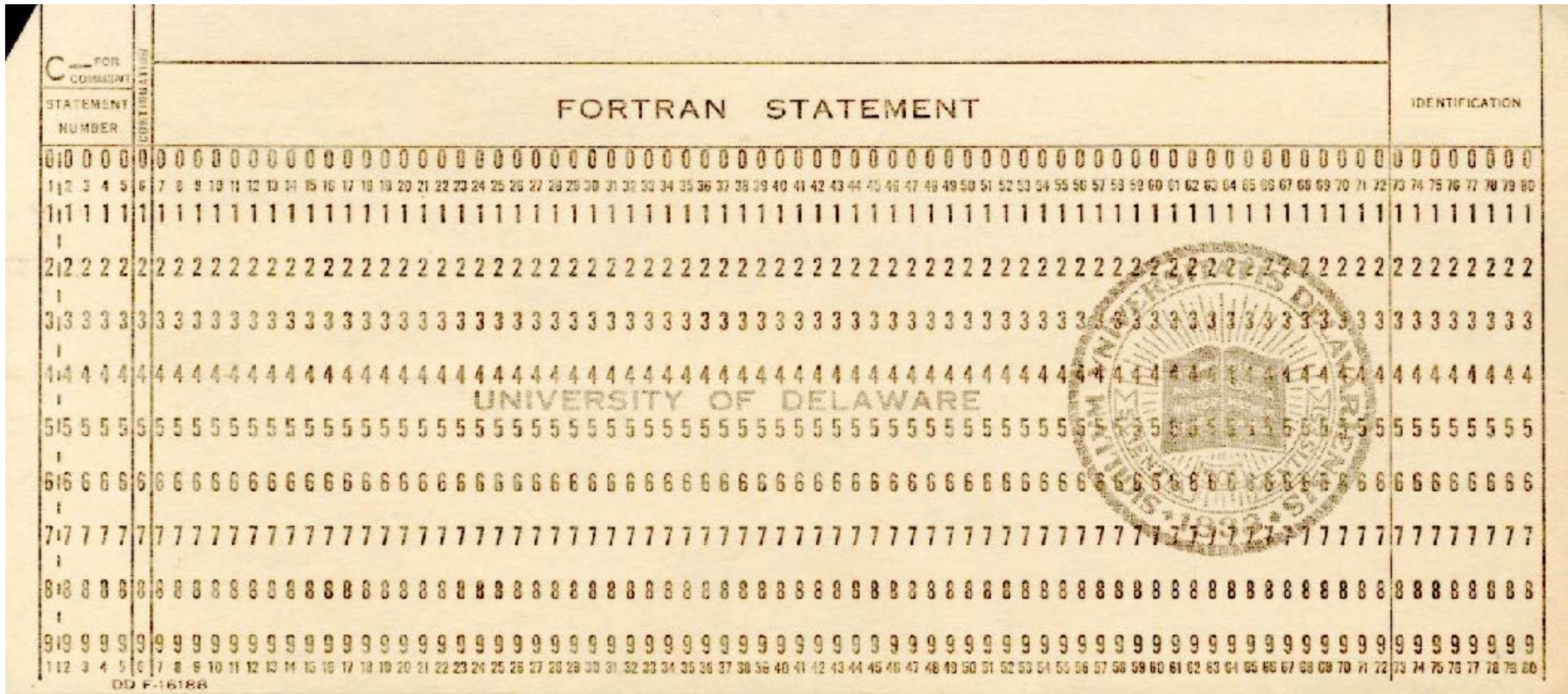
```
1010 1001 0000 0011
1000 1101 0100 0001 0000 0000
```

```
A9 03
8D 41 00
```

But how could you store it?

A long time ago....

The first storage



(Taken from Chris Algozzine's office.)

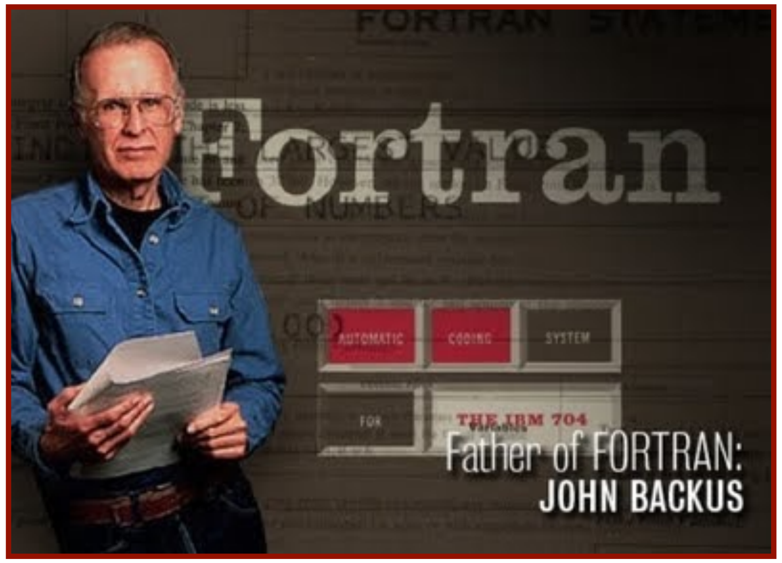
(Written by Bill Gates and Paul Allen)

Now the pieces are in place . . .

A long time ago….

FORTRAN

- **FOR**mula **TRAN**slator
- for Scientific computing
- by John Backus from IBM



COBOL

- **CO**mmon **B**usiness **O**riented **L**anguage
- 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	int limit = 3;	var limit = 3;
20 X = 1	int val = 1;	var val = 1;
30 print X	repeat	do
40 X = X + 1	{	{
50 IF X <> A THEN GOTO 30	console.write(val);	alert(val);
60 PRINT “DONE”	++val;	++val;
	} until (val == limit)	} while (val <> limit);
	console.write(“DONE”);	alert(“DONE”);

What languages are these?

Before and After Compilation

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

lda #$3	*Acc = 3*	0000	LDA #$03	A9 03
sta $0041	*Mem[41] = 3*	0002	STA $0041	8D 41 00
lda #$1	*Acc = 1*	0005	LDA #$01	A9 01
sta $0040	*Mem[40] = 1*	0007	STA $0040	8D 40 00
loop ldy $0040	*Y = Mem[40]*	000A **LOOP**	LDY $0040	AC 40 00
ldx #$01	*X = 1*	000D	LDX #$01	A2 01
sys	*System Call*	000F	SYS	FF
inc $0040	*Mem[40]++*	0010	INC $0040	EE 40 00
ldx $0040	*X = Mem[40]*	0013	LDX $0040	AE 40 00
cpx $0041	*Z bit = (x == Mem[41])*	0016	CPX $0041	EC 41 00
bne loop	*if z == 0 goto loop*	0019	BNE **LOOP**	D0 EF
lda #$44	*Acc = $44 ("D")*	001B	LDA #$44	A9 44
sta $0042	*Mem[42] = $44*	001D	STA $0042	8D 42 00
lda #$4F	*Acc = $4F ("O")*	0020	LDA #$4F	A9 4F
sta $0043	*Mem[43] = $4F*	0022	STA $0043	8D 43 00
lda #$4E	*Acc = $4E ("N")*	0025	LDA #$4E	A9 4E
sta $0044	*Mem[44] = $4E*	0027	STA $0044	8D 44 00
lda #$45	*Acc = $45 ("E")*	002A	LDA #$45	A9 45
sta $0045	*Mem[45] = $45*	002C	STA $0045	8D 45 00
lda #$00	*Acc = $00 (null)*	002F	LDA #$00	A9 00
sta $0046	*Mem[46] = $00*	0031	STA $0046	8D 46 00
ldx #$02	*X = 2*	0034	LDX #$02	A2 02
ldy #$42	*Y = $42 (address)*	0036	LDY #$42	A0 42
sys	*System call*	0038	SYS	FF
brk	*Break*	0039	BRK	00

Before and After Compilation

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

lda #$3	*Acc = 3*	0000	LDA #$03	A9 03
sta $0041	*Mem[41] = 3*	0002	STA $0041	8D 41 00
lda #$1	*Acc = 1*			
sta $0040	*Mem[40] = 1*			
loop ldy $0040	*Y = Mem[40]*			
ldx #$01	*X = 1*			
sys	*System Call*	000F	SYS	FF
inc $0040	*Mem[40]++*	0010	INC $0040	EE 40 00
ldx $0040	*X = Mem[40]*	0013	LDX $0040	AE 40 00
cpx $0041	*Z bit = (x == Mem[41])*	0016	CPX $0041	EC 41 00
bne loop	*if z == 0 goto loop*	0019	BNE **LOOP**	D0 EF
lda #$44	*Acc = $44 ("D")*	001B	LDA #$44	A9 44
sta $0042	*Mem[42] = $44*	001D	STA $0042	8D 42 00
lda #$4F	*Acc = $4F ("O")*	0020	LDA #$4F	A9 4F
sta $0043	*Mem[43] = $4F*	0022	STA $0043	8D 43 00
lda #$4E	*Acc = $4E ("N")*	0025	LDA #$4E	A9 4E
sta $0044	*Mem[44] = $4E*	0027	STA $0044	8D 44 00
lda #$45	*Acc = $45 ("E")*	002A	LDA #$45	A9 45
sta $0045	*Mem[45] = $45*	002C	STA $0045	8D 45 00
lda #$00	*Acc = $00 (null)*	002F	LDA #$00	A9 00
sta $0046	*Mem[46] = $00*	0031	STA $0046	8D 46 00
ldx #$02	*X = 2*	0034	LDX #$02	A2 02
ldy #$42	*Y = $42 (address)*	0036	LDY #$42	A0 42
sys	*System call*	0038	SYS	FF
brk	*Break*	0039	BRK	00

1010 1001 0000 0011

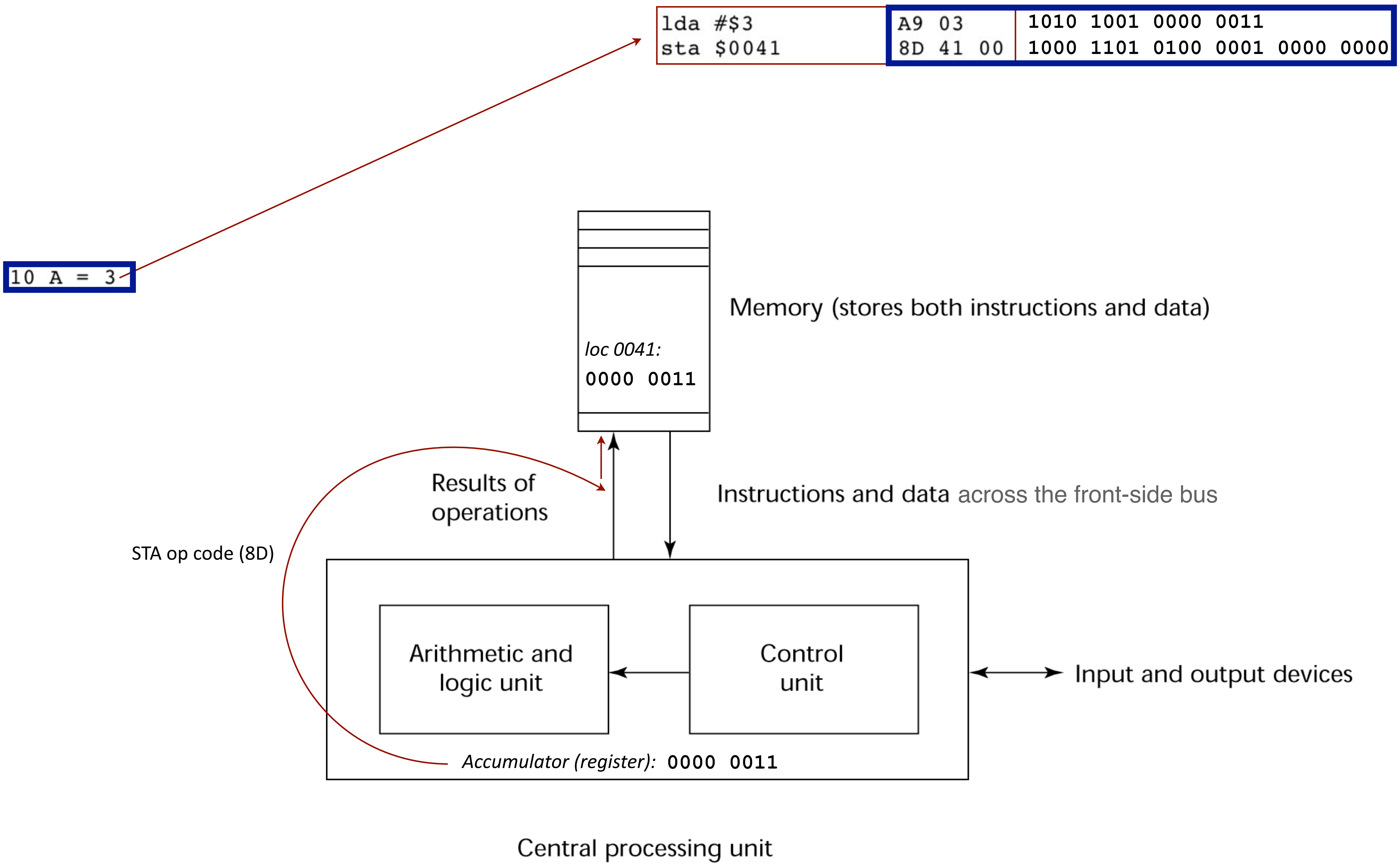
1000 1101 0100 0001 0000 0000

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             A9 03         1010 1001 0000 0011
sta $0041           8D 41 00      1000 1101 0100 0001 0000 0000

lda #$1             A9 01         1010 1001 0000 0001
sta $0040           8D 40 00      1000 1101 0100 0000 0000 0000

loop ldy $0040      AC 40 00      1010 1100 0100 0000 0000 0000
     ldx #$01       A2 01         1010 0010 0000 0001
     sys            FF            1111 1111

     inc $0040      EE 40 00            .
     ldx $0040      AE 40 00            .
                                        .
     cpx $0041      EC 41 00
     bne loop       D0 EF

lda #$44            A9 44
sta $0042           8D 42 00
lda #$4F            A9 4F
sta $0043           8D 43 00
lda #$4E            A9 4E
sta $0044           8D 44 00
lda #$45            A9 45
sta $0045           8D 45 00
lda #$00            A9 00
sta $0046           8D 46 00
ldx #$02            A2 02
ldy #$42            A0 42
sys                 FF

brk                 00
```

Aside: von Neumann Architecture



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             A9 03         1010 1001 0000 0011
sta $0041           8D 41 00      1000 1101 0100 0001 0000 0000

lda #$1             A9 01         1010 1001 0000 0001
sta $0040           8D 40 00      1000 1101 0100 0000 0000 0000

loop ldy $0040      AC 40 00      1010 1100 0100 0000 0000 0000
     ldx #$01       A2 01         1010 0010 0000 0001
     sys            FF            1111 1111

     inc $0040      EE 40 00           .
     ldx $0040      AE 40 00           .
                                       .
     cpx $0041      EC 41 00
     bne loop       D0 EF

lda #$44            A9 44
sta $0042           8D 42 00
lda #$4F            A9 4F
sta $0043           8D 43 00
lda #$4E            A9 4E
sta $0044           8D 44 00
lda #$45            A9 45
sta $0045           8D 45 00
lda #$00            A9 00
sta $0046           8D 46 00
ldx #$02            A2 02
ldy #$42            A0 42
sys                 FF

brk                 00
```

Before and After Compilation

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 A9 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
```

How are we going to execute this?

Before and After Compilation

Source Code

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;
		do
		{
		  alert(val);
		  ++val;
		} while (val <> limit);
		alert("DONE");

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

lda #$3	*Acc = 3*	0000		LDA #$03	A9 03
sta $0041	*Mem[41] = 3*	0002		STA $0041	8D 41 00
lda #$1	*Acc = 1*	0005		LDA #$01	A9 01
sta $0040	*Mem[40] = 1*	0007		STA $0040	8D 40 00
loop ldy $0040	*Y = Mem[40]*	000A	**LOOP**	LDY $0040	AC 40 00
ldx #$01	*X = 1*	000D		LDX #$01	A2 01
sys	*System Call*	000F		SYS	FF
inc $0040	*Mem[40]++*	0010		INC $0040	EE 40 00
ldx $0040	*X = Mem[40]*	0013		LDX $0040	AE 40 00
cpx $0041	*Z bit = (x == Mem[41])*	0016		CPX $0041	EC 41 00
bne loop	*if z == 0 goto loop*	0019		BNE **LOOP**	D0 EF
lda #$44	*Acc = $44 ("D")*	001B		LDA #$44	A9 44
sta $0042	*Mem[42] = $44*	001D		STA $0042	8D 42 00
lda #$4F	*Acc = $4F ("O")*	0020		LDA #$4F	A9 4F
sta $0043	*Mem[43] = $4F*	0022		STA $0043	8D 43 00
lda #$4E	*Acc = $4E ("N")*	0025		LDA #$4E	A9 4E
sta $0044	*Mem[44] = $4E*	0027		STA $0044	8D 44 00
lda #$45	*Acc = $45 ("E")*	002A		LDA #$45	A9 45
sta $0045	*Mem[45] = $45*	002C		STA $0045	8D 45 00
lda #$00	*Acc = $00 (null)*	002F		LDA #$00	A9 00
sta $0046	*Mem[46] = $00*	0031		STA $0046	8D 46 00
ldx #$02	*X = 2*	0034		LDX #$02	A2 02
ldy #$42	*Y = $42 (address)*	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 A9 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
```

Machine Language

Super-High Level View

Source Code

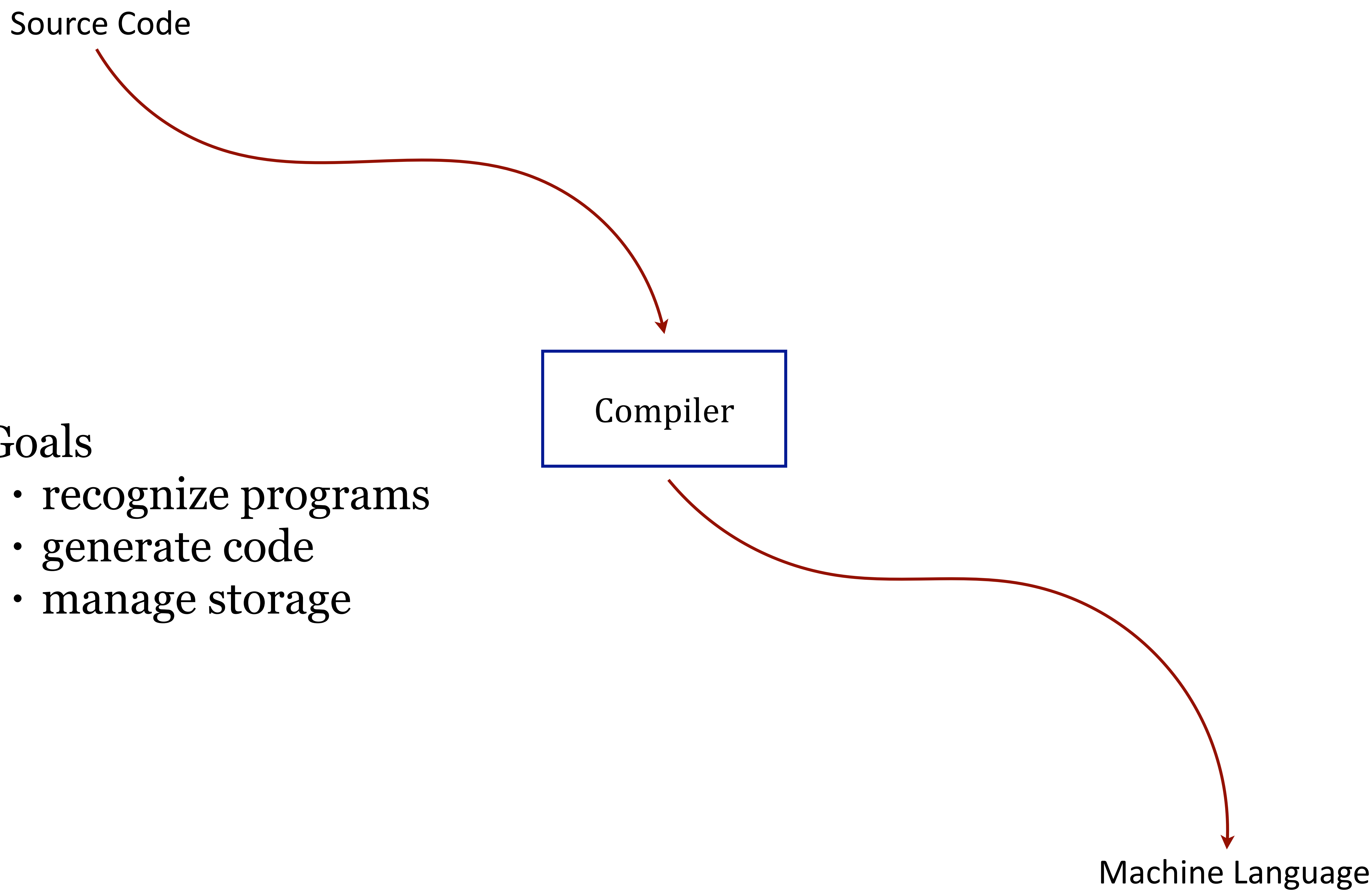
↓

Translator?

↓

Machine Language

Super-High Level View

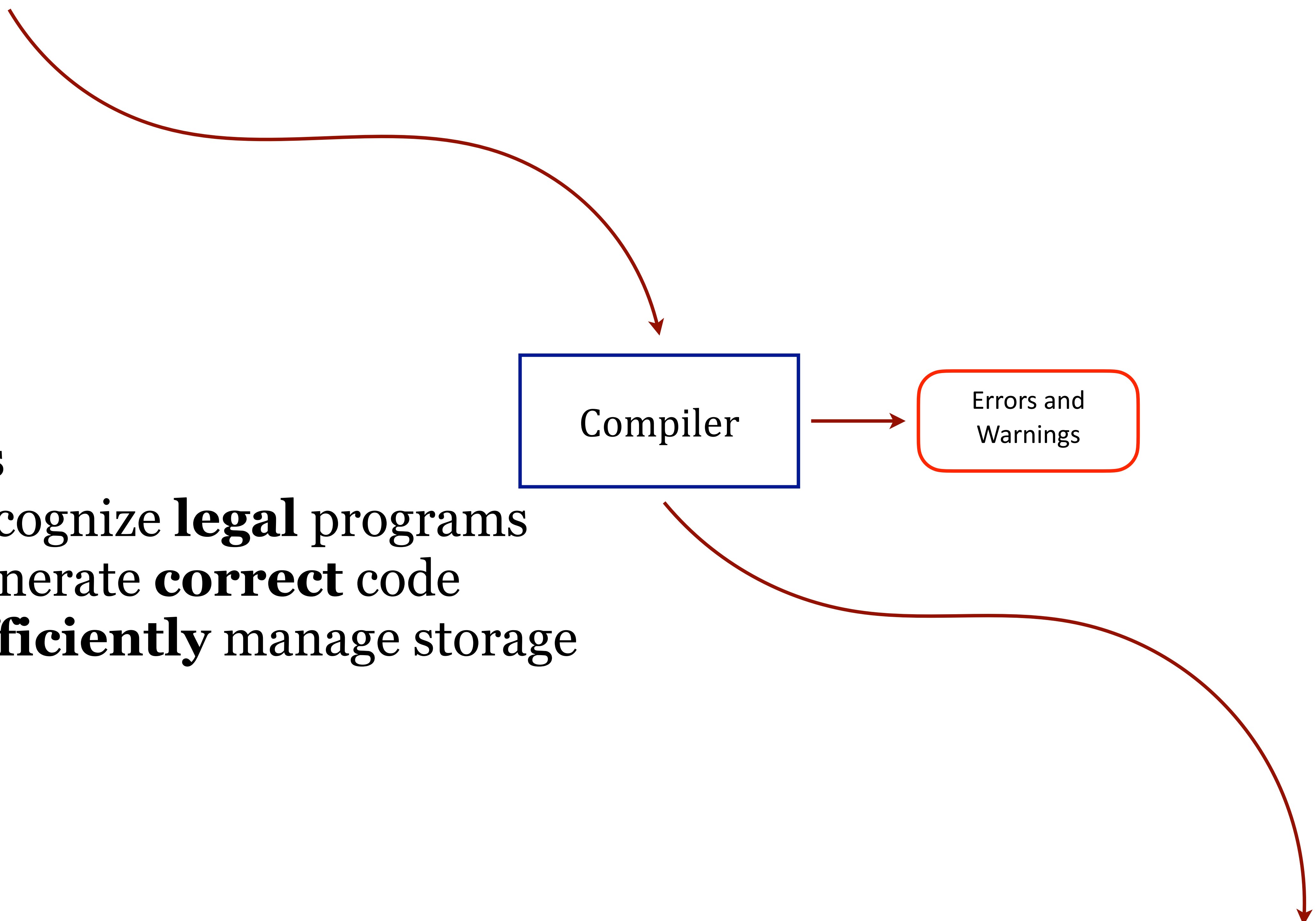


Goals

- recognize programs
- generate code
- manage storage

Super-High Level View

Source Code



Goals

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

Machine Language

Super-High Level View

Source Code

Front End

Errors and Warnings

Intermediate Representation

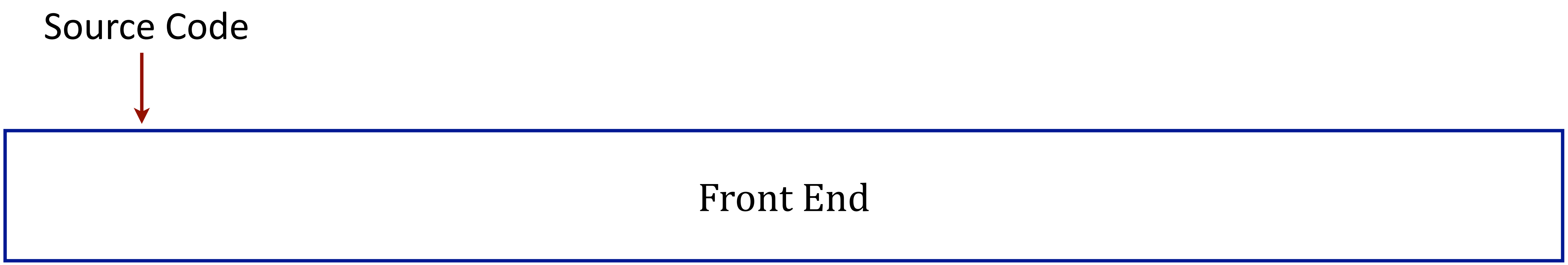
Goals

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

Back End

Machine Language

High Level View: Front End

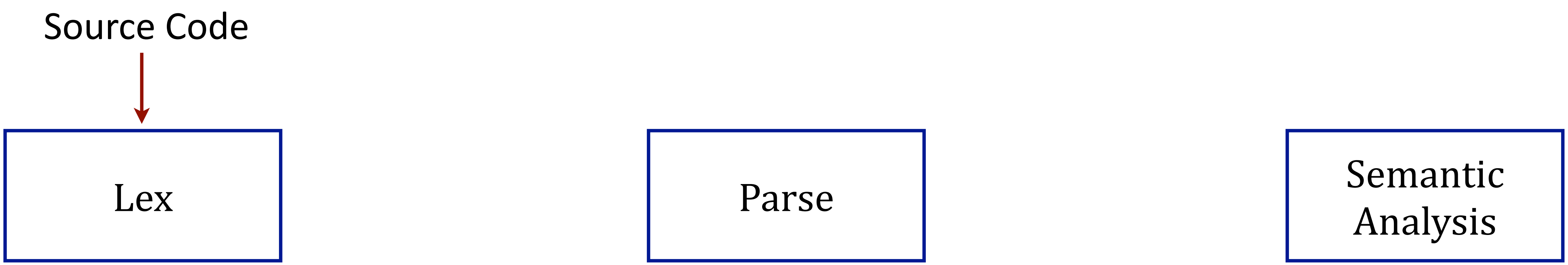


Front End

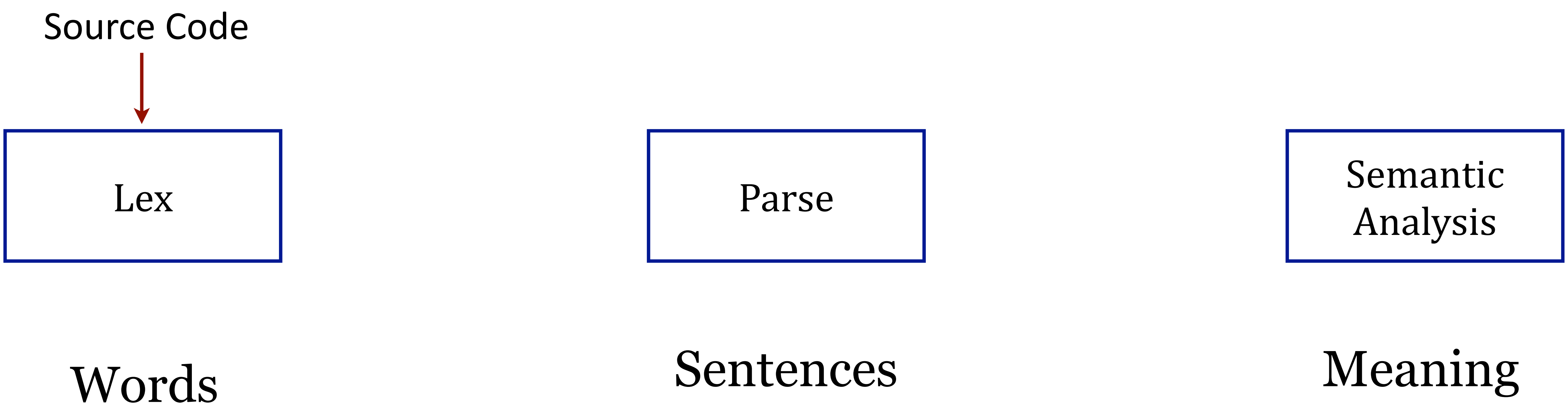
- Recognize legal and illegal code
- Keep track of variables and types
- Report errors and warnings
- Produce the Intermediate Representation (IR)

- Consists of three parts:

High Level View: Front End



High Level View: Front End



High Level View: Front End

Source Code

Lex

Parse

Semantic Analysis

Words

Sentences

Meaning

GLaDOS
~~cke~~
cake
ate

High Level View: Front End

Source Code

↓

Lex | Parse | Semantic Analysis

Words

```
GLaDOS
```
~~cke~~
```
cake
ate
```

Sentences

~~ate cake GLaDOS~~

```
GLaDOS ate cake
```

Meaning

High Level View: Front End

Source Code

↓

Lex

Parse

Semantic Analysis

Words

`GLaDOS`

~~`cke`~~

`cake`

`ate`

Sentences

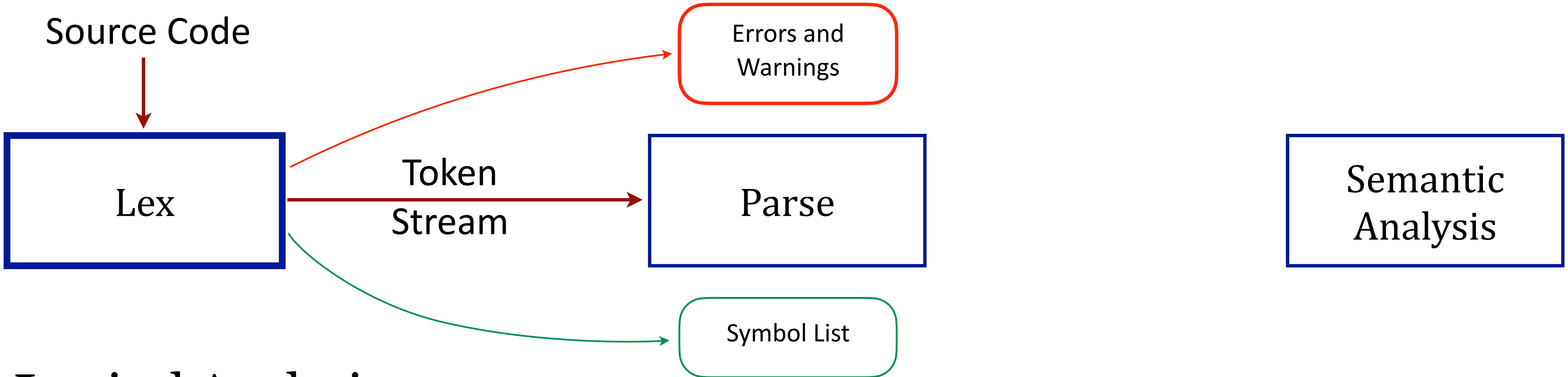
~~`ate cake GLaDOS`~~

`GLaDOS ate cake`

Meaning

`GLaDOS`	`ate`	`cake`
noun	*verb*	*direct object*

High Level View: Lex



Lexical Analysis

- Maps characters into an ordered stream of tokens

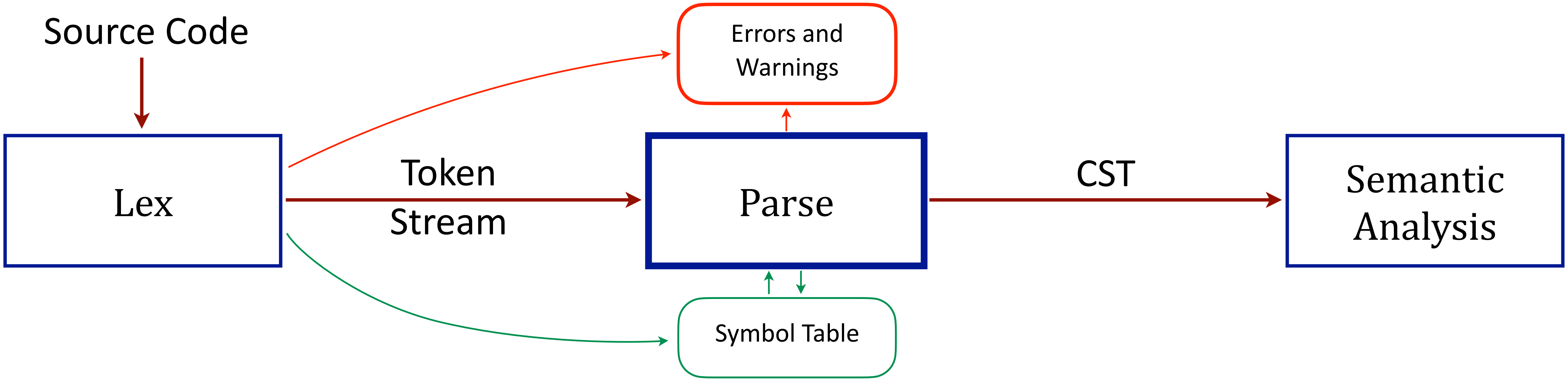
  `x := x + y`

  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

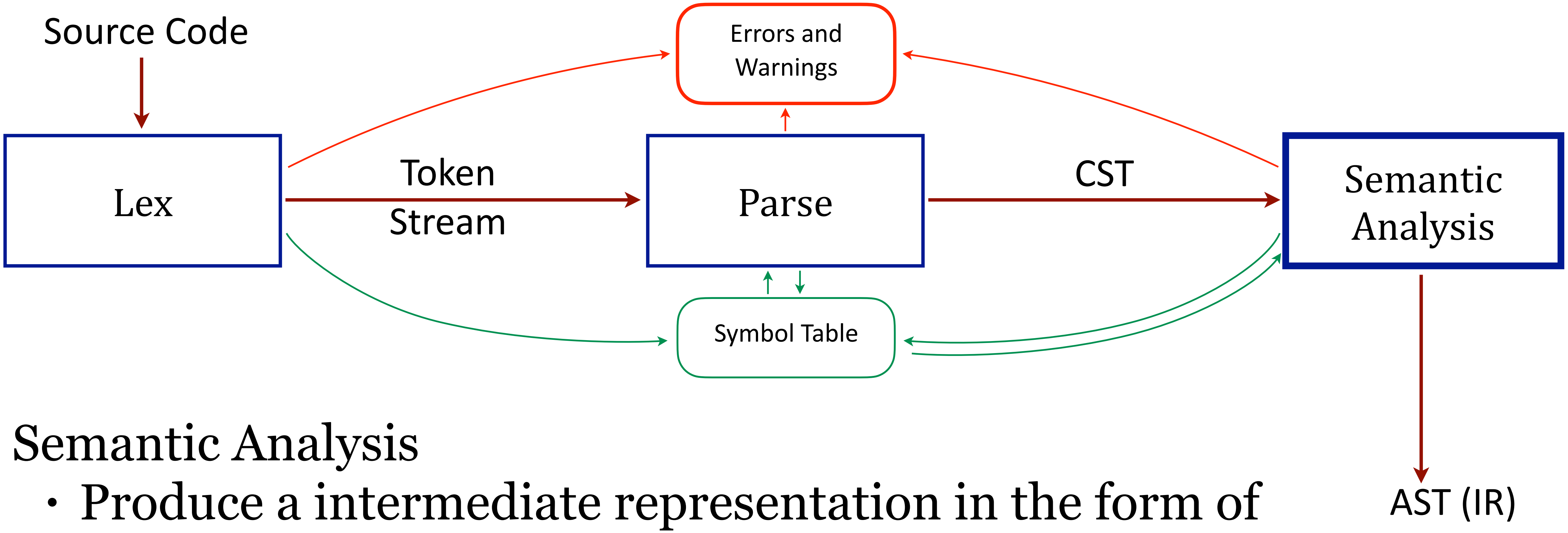


Parse

- 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

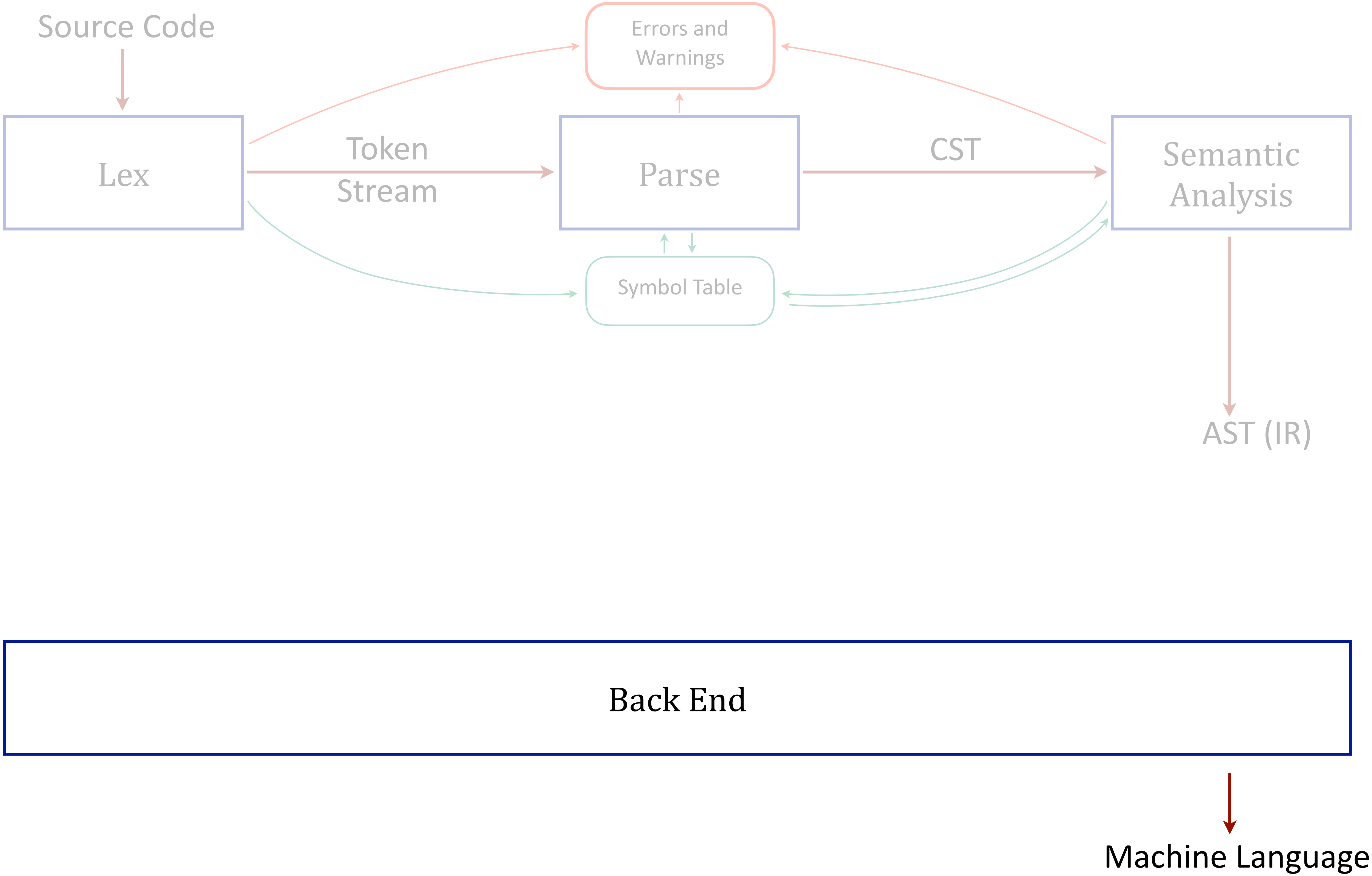


Semantic Analysis

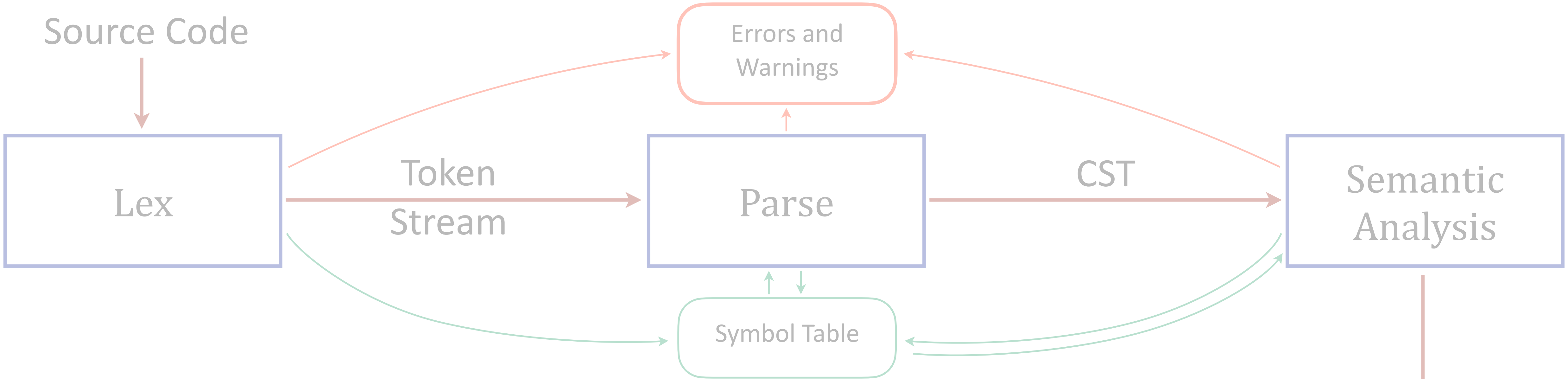
- Produce a intermediate representation in the form of an Abstract Syntax Tree (AST)
- Check type
- Check scope
- Report meaningful errors and warnings

- Focus on meaning

High Level View: Back End

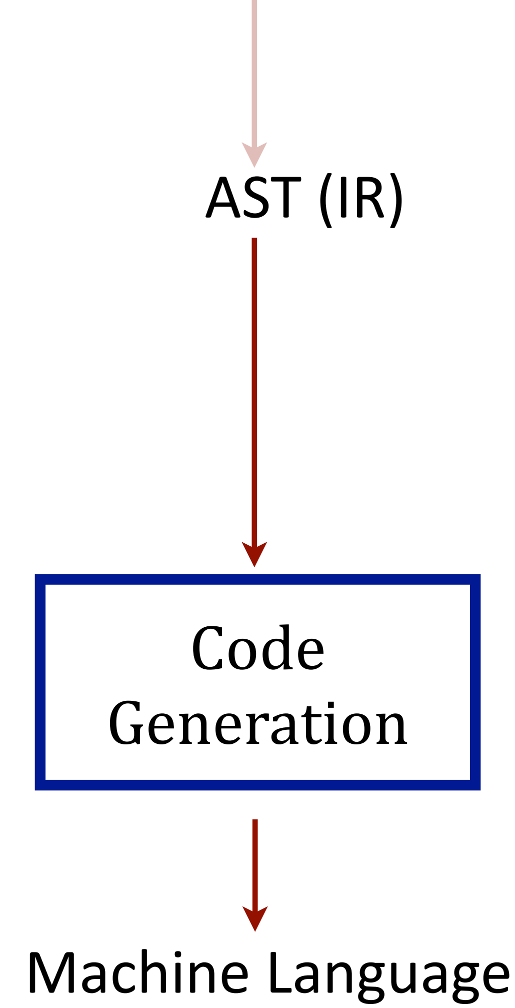


High Level View: Back End

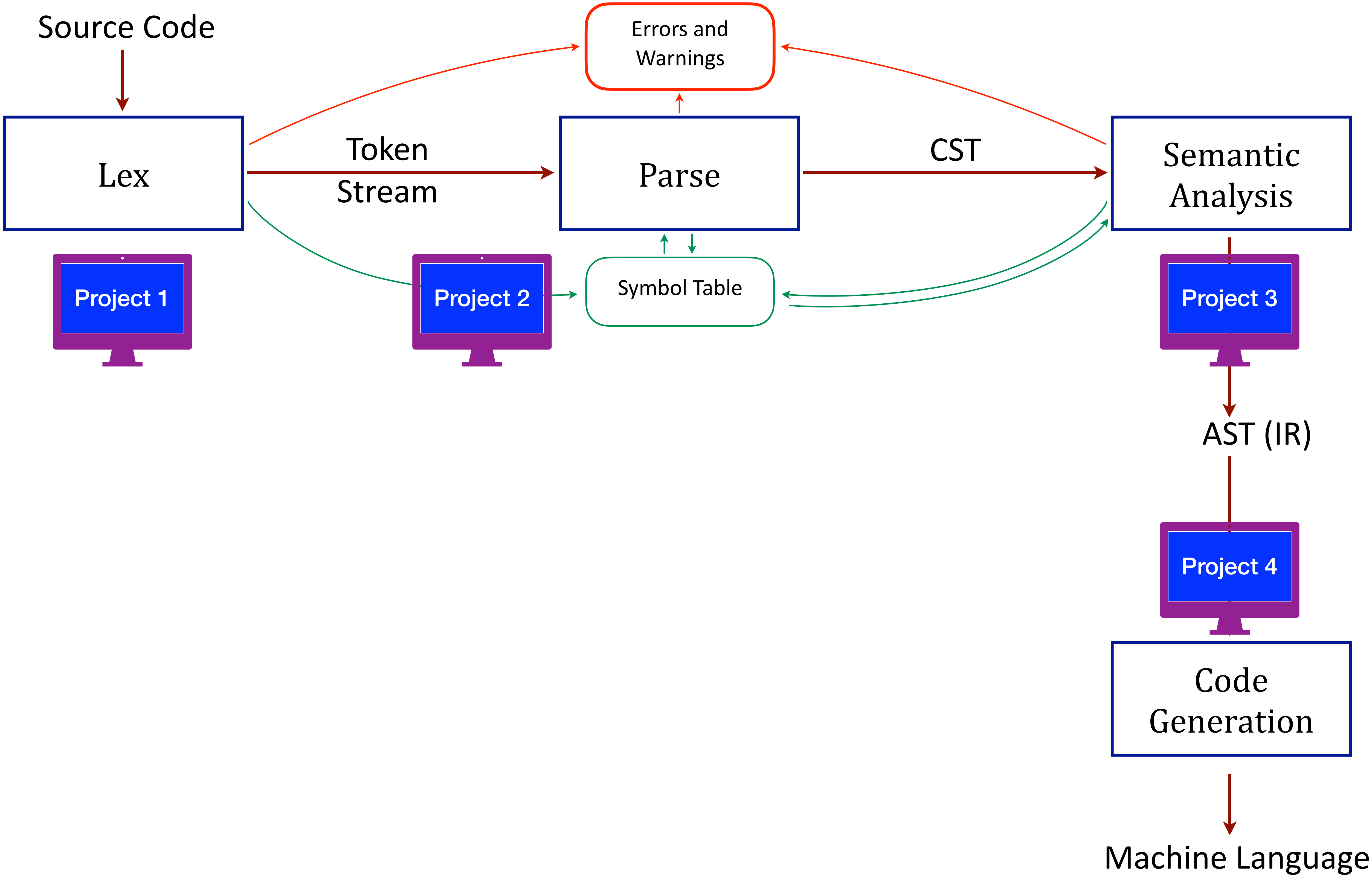


Code Generation

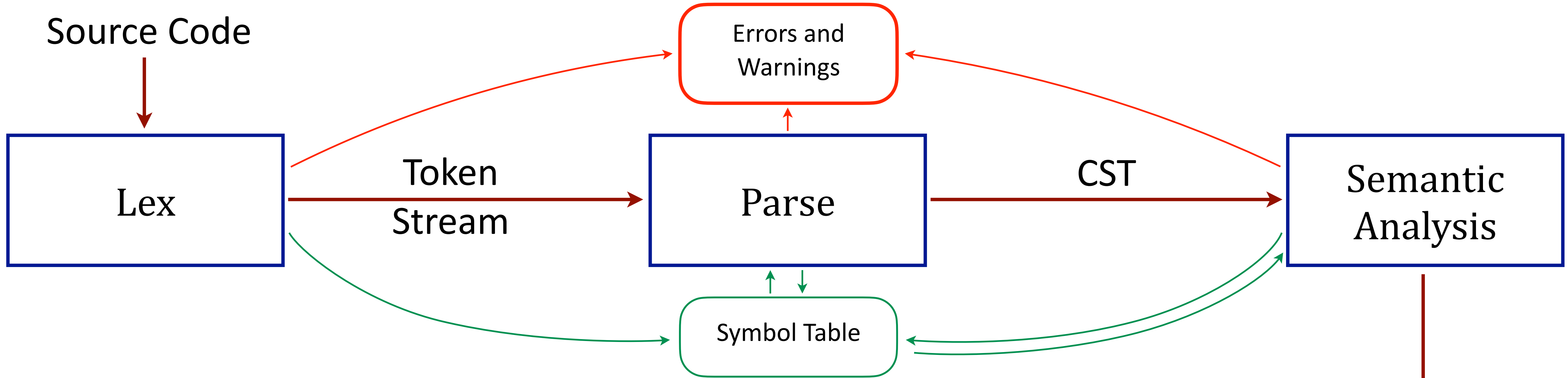
- Translates IR into Machine Language for the target platform
- Chose instructions for each IR operation
- Allocate registers
- Ensure conformance with target system



High Level View



High Level View: Back End



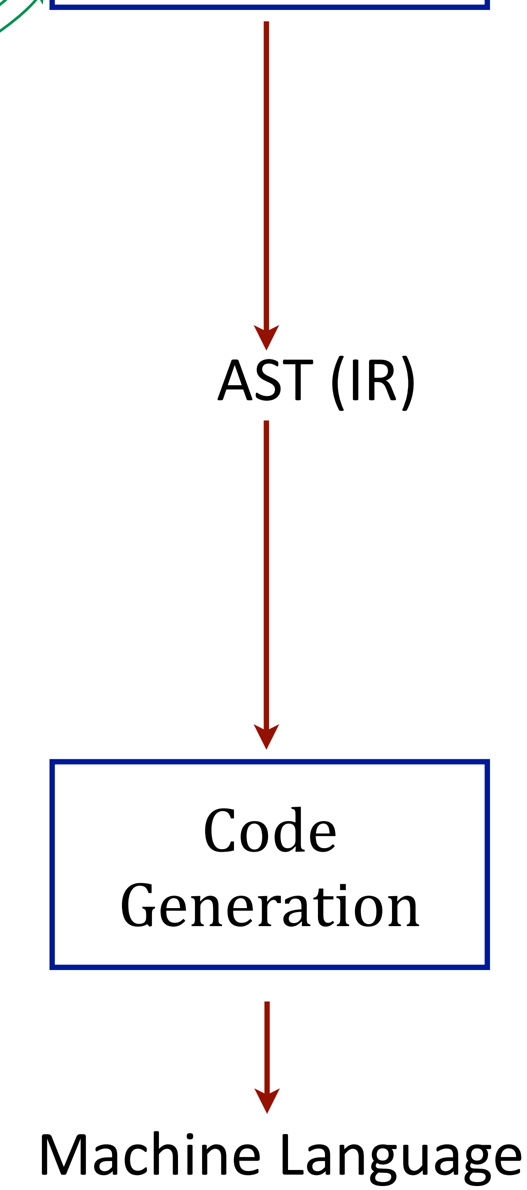
There's more to the back end of a compiler:

- Lots of AST optimizations
- Instruction Selection
- Lots of instruction optimizations
- Register Allocation

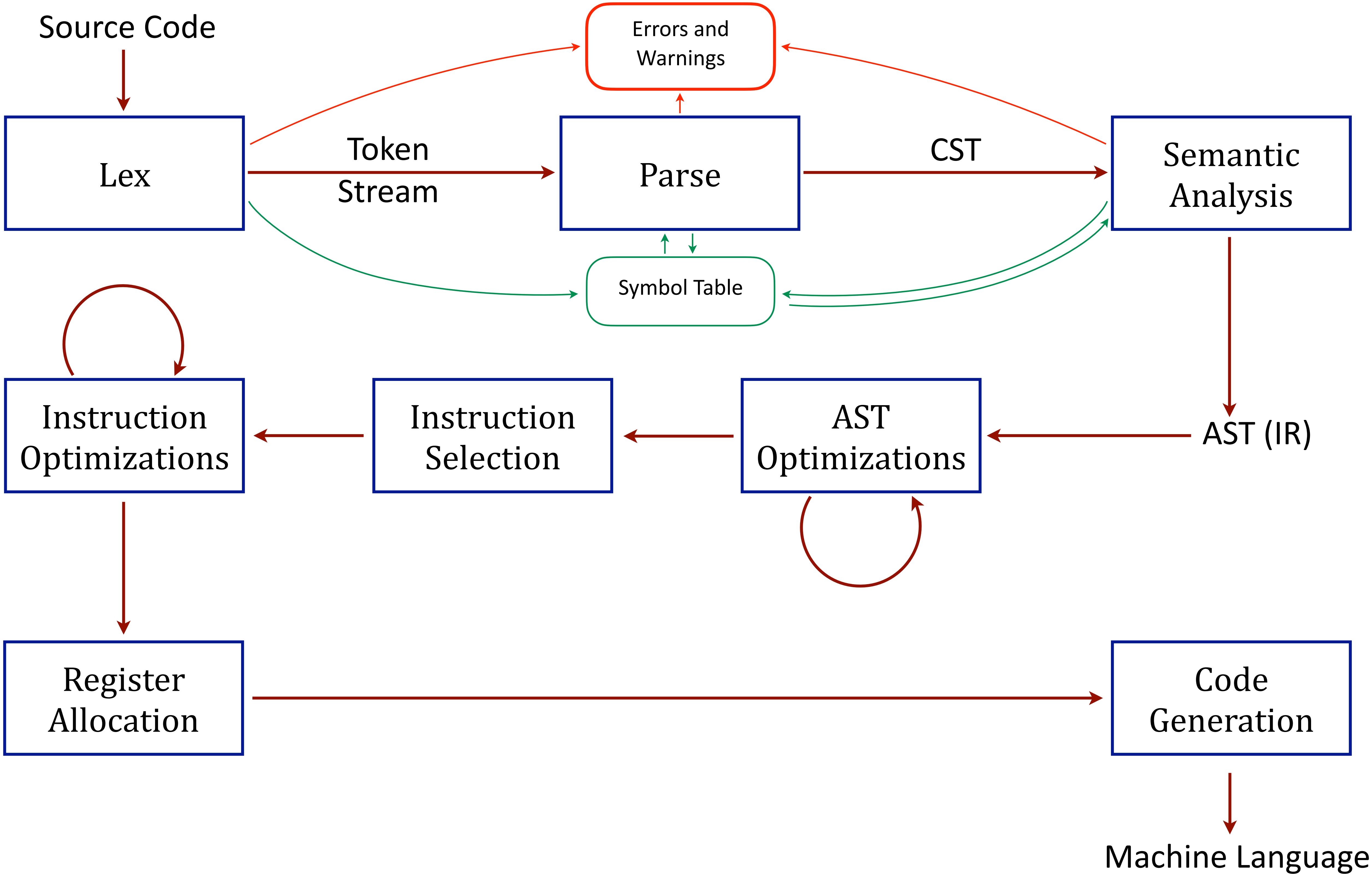
Sadly, we don't have time to implement these along with everything else in a single semester. But here's a peek…

Dead code elimination
Global CSE
Code motion
Constant folding
Strength reduction
Value numbering
Dead store elimination
Code straightening
Trap elimination
Algebraic reassociation

From Cooper & Torczon



High Level View: Back End



I wonder . . .

Source Code

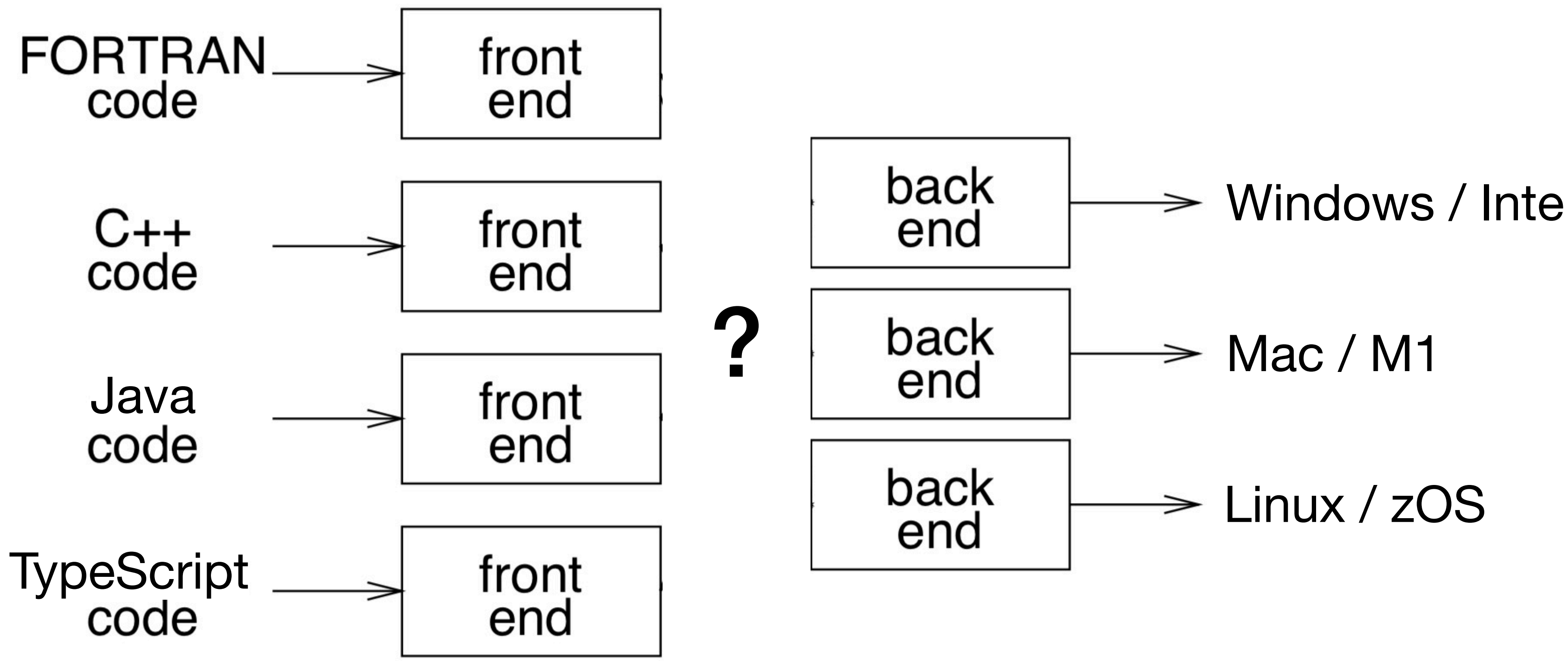
Front End

How abstract should we get here?

Back End

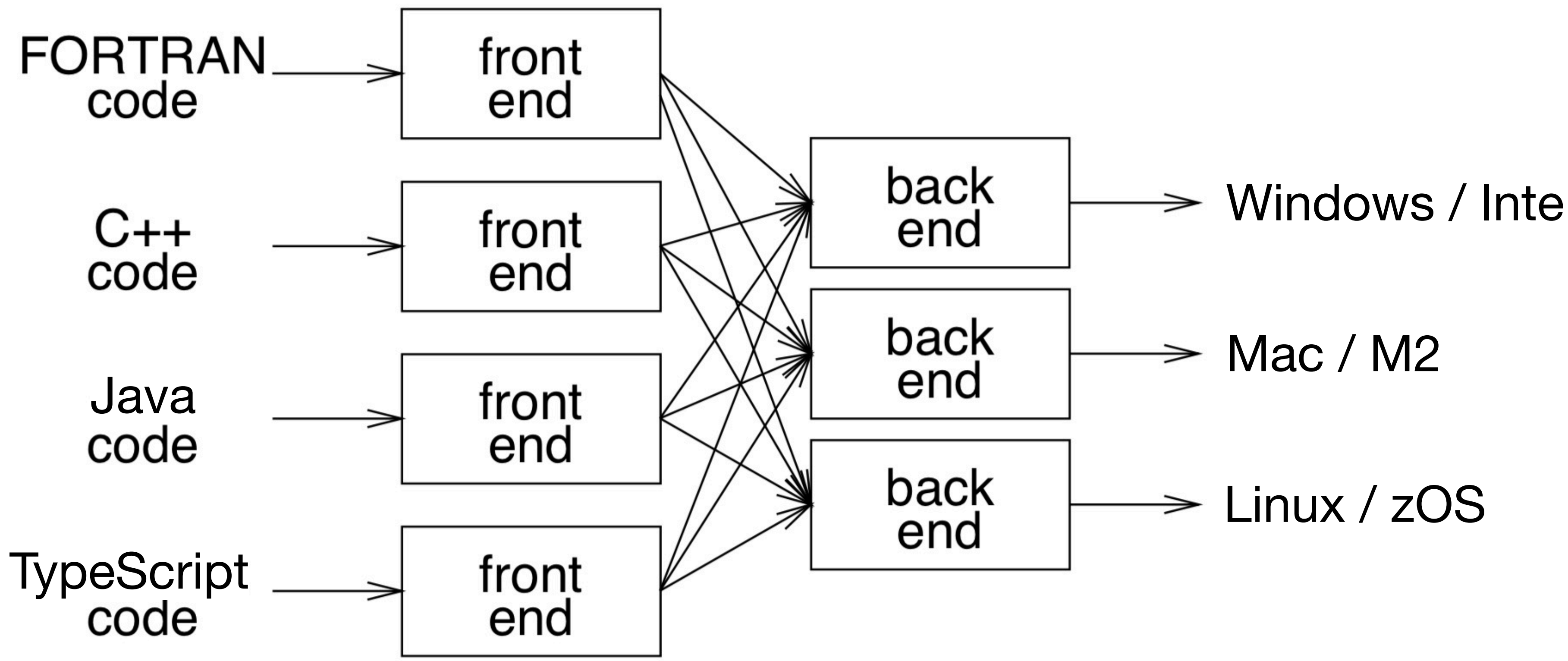
Machine Language

I wonder . . .



Can we build $n \times m$ compilers with $n + m$ components?

I wonder . . .



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

Example

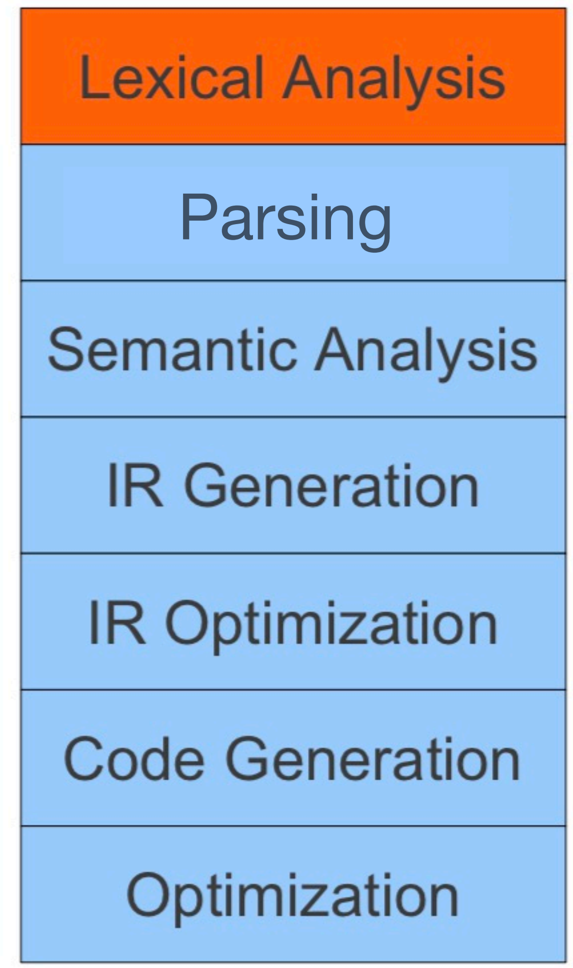
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

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

Example

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

```
While [ while ] found at (6:1)
LParen [ ( ] found at (6:9)
Id [ y ] found at (6:10)
BoolOp [ < ] found at (6:12)
Id [ z ] found at (6:15)
RParen [ ) ] found at (6:16)
LBrace [ { ] found at (6:18)
TypeDef [ int ] found at (7:4)
Id [ x ] found at (7:10)
Id [ x ] found at (8:6)
Assign [ = ] found at (8:8)
Id [ a ] found at (8:10)
IntOp [ + ] found at (8:12)
Id [ b ] found at (8:14)
Semicolon found at (8:15)
Id [ y ] found at (9:6)
Assign [ = ] found at (9:8)
Id [ y ] found at (9:10)
IntOp [ + ] found at (9:12)
Id [ x ] found at (9:14)
Semicolon found at (9:15)
RBrace [ } ] found at (10:3)
Lexical Analysis complete with 0 WARNING(S) and 0 ERROR(S)
```



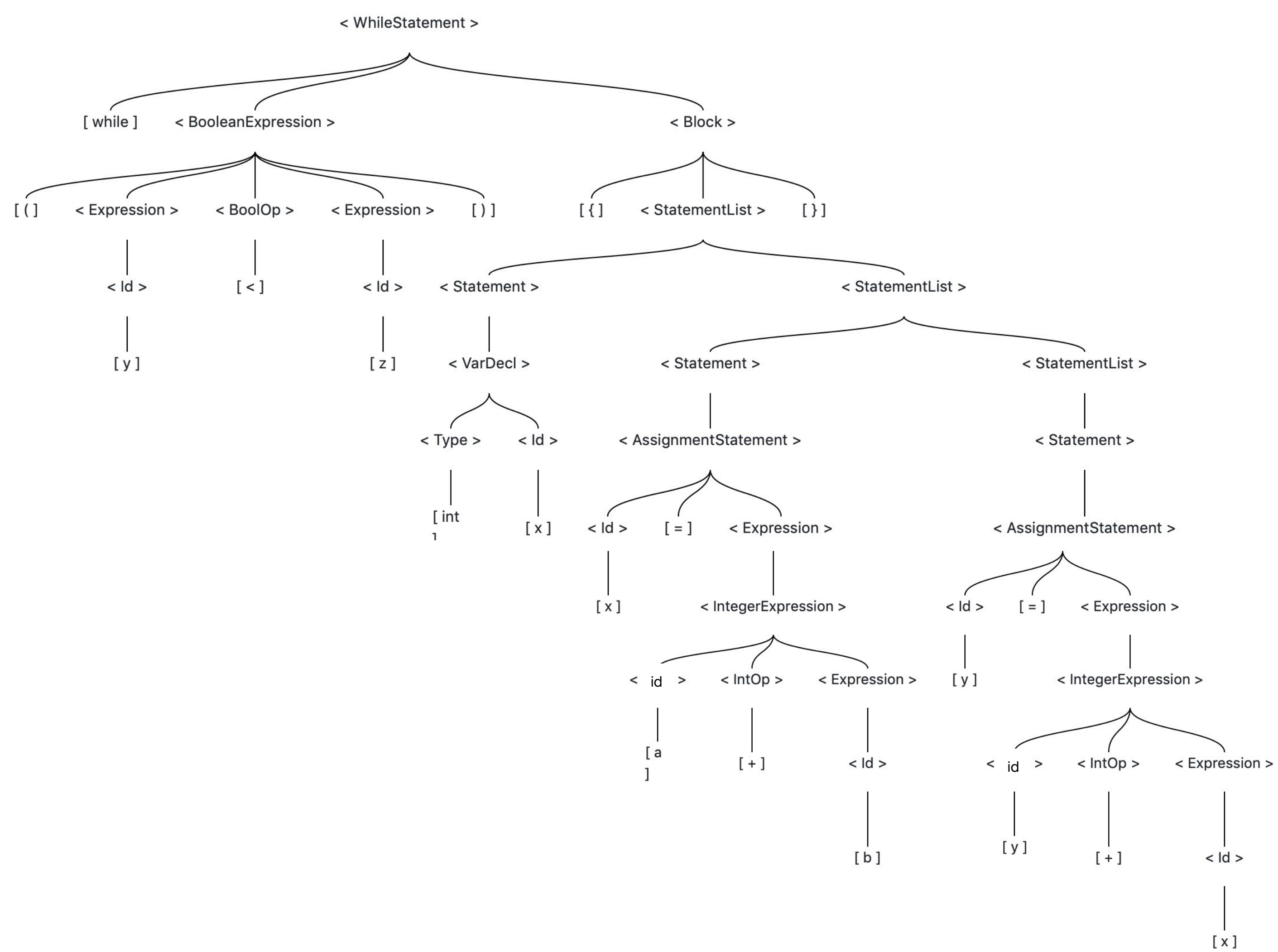
Tokens

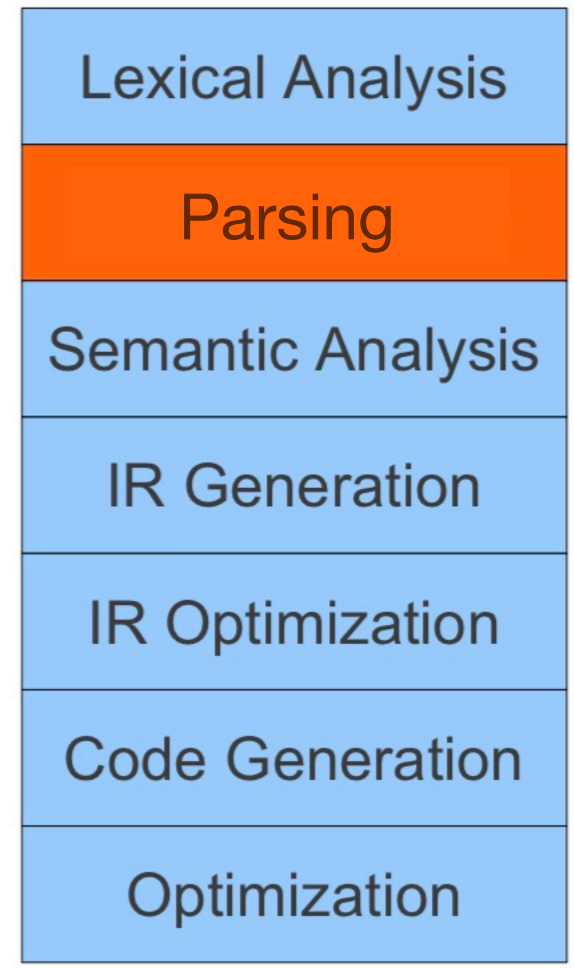
Lex: Did we get the **words** right?

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

Example

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





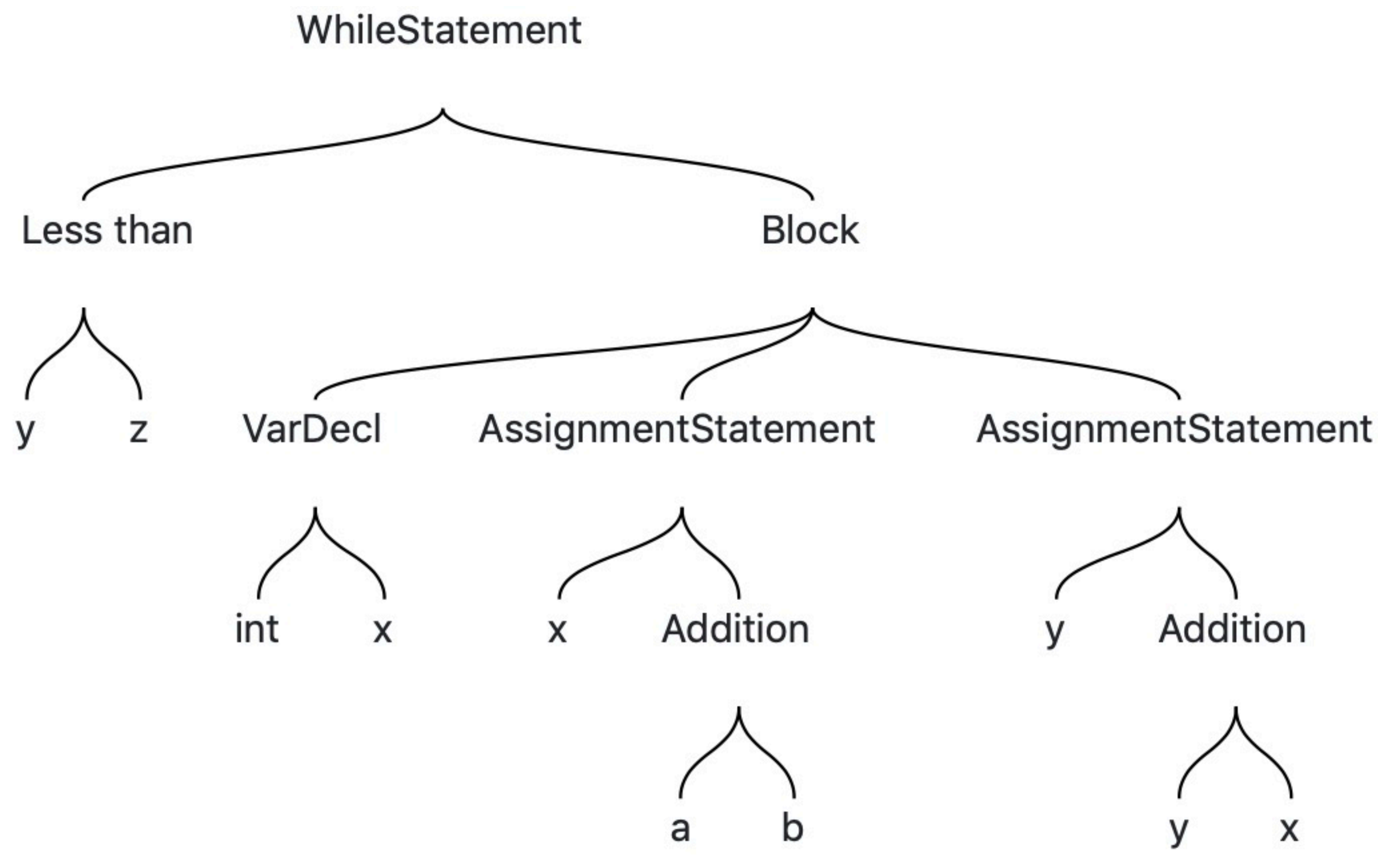
CST

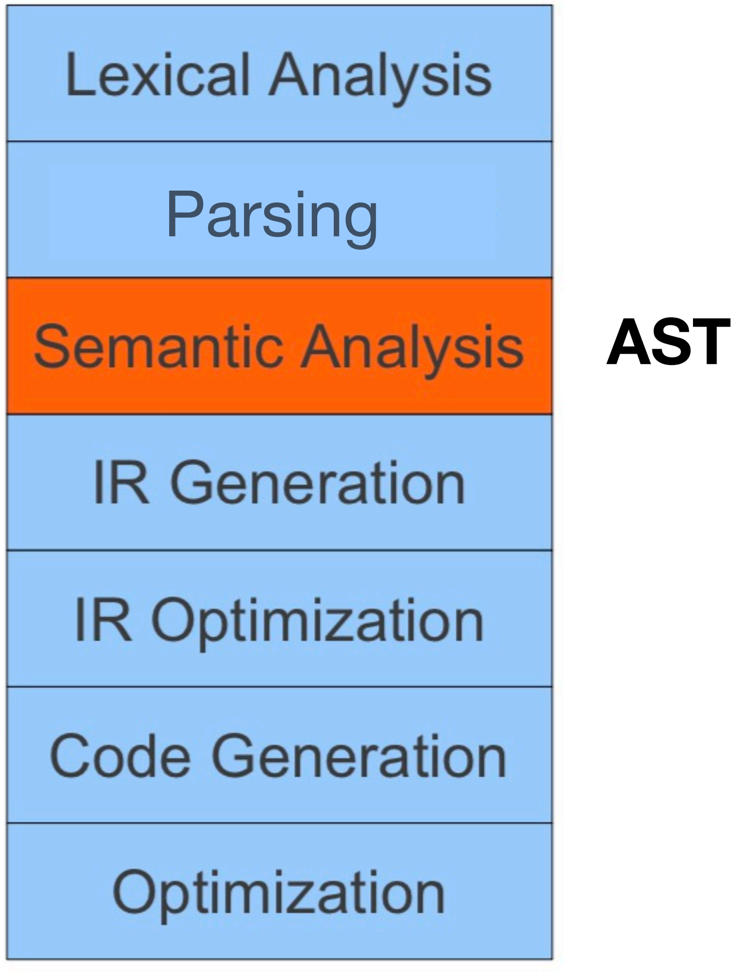
Parse: Did we get the **sentences** right?

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.





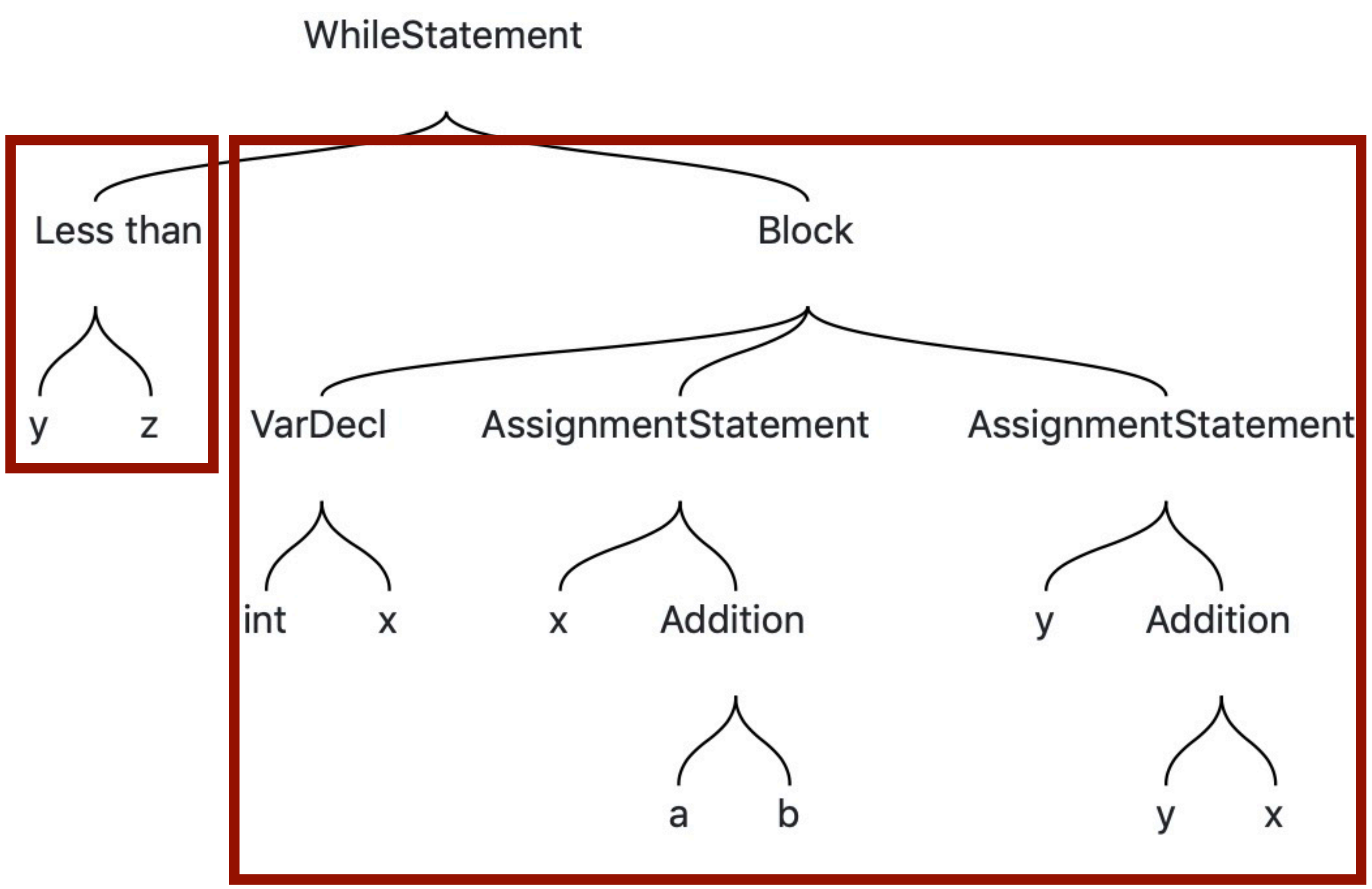
SA: Did we get the **meaning** (scope and type) right?

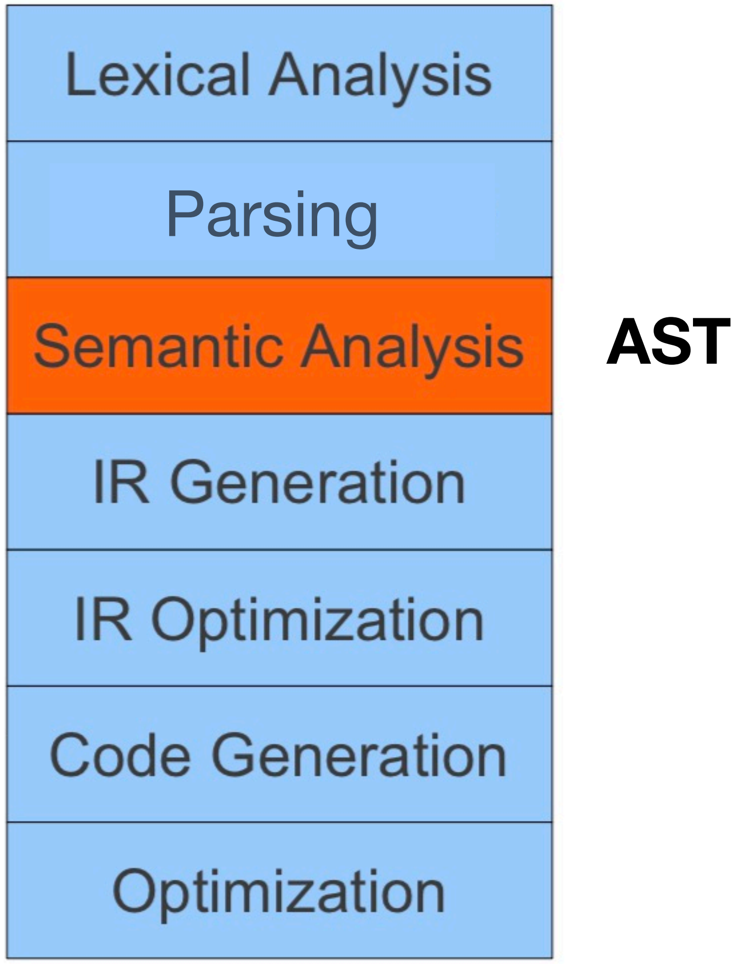
This is based on an example from the great Alex Aiken.

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.





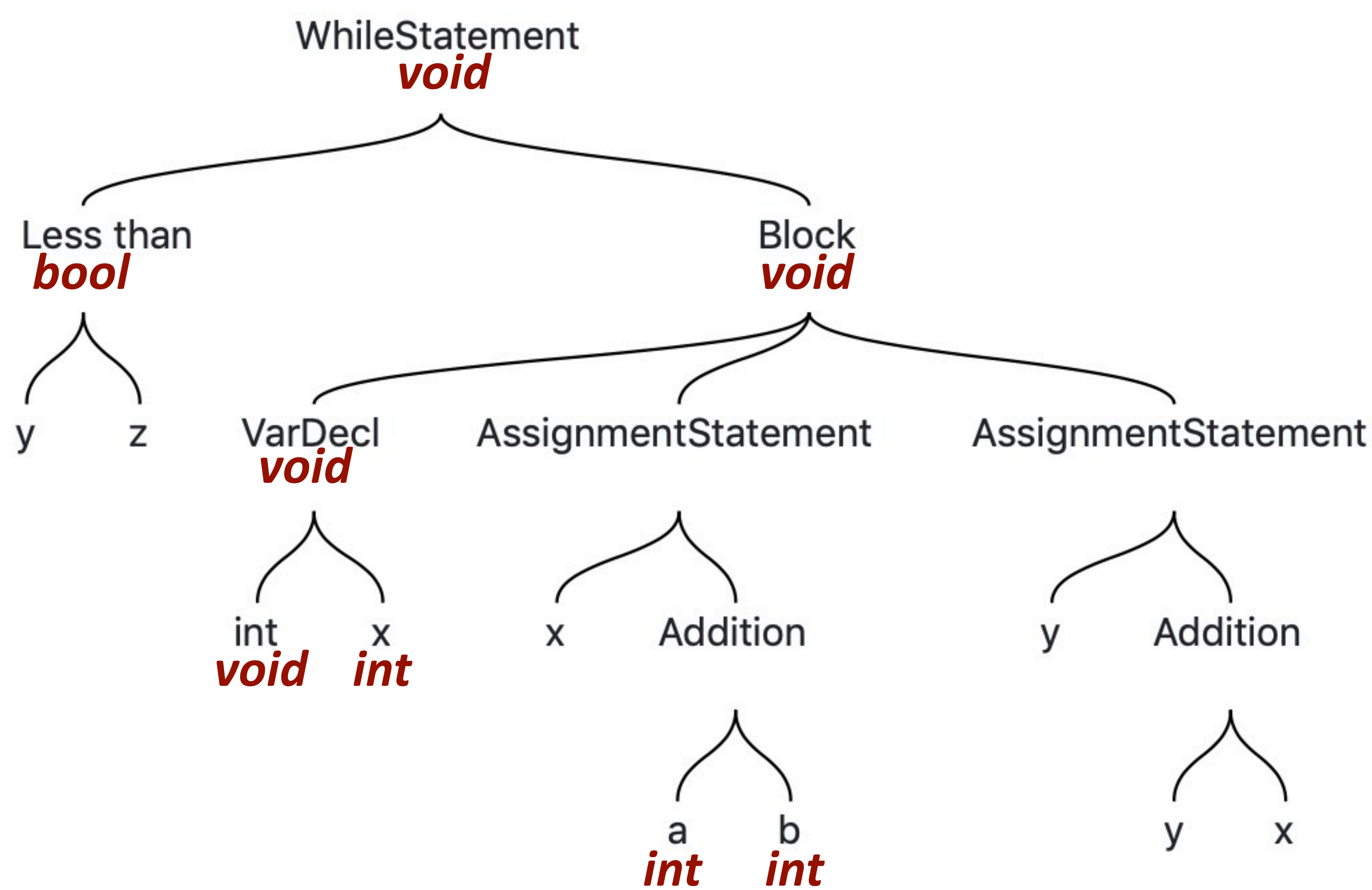
SA: Did we get the **meaning** (**scope** and type) right?

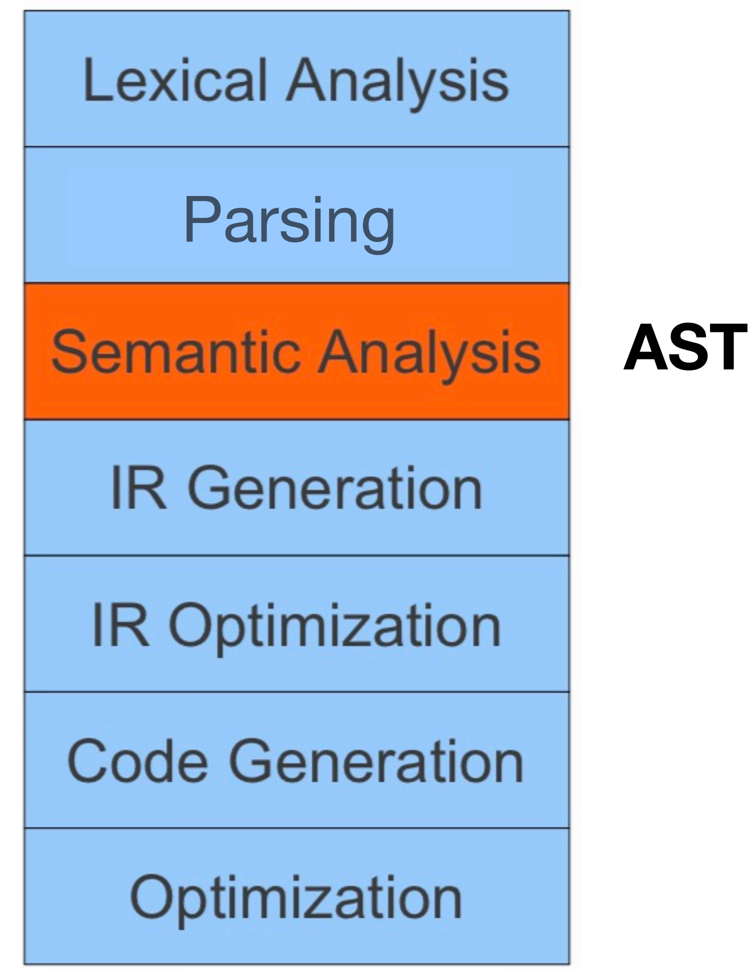
This is based on an example from the great Alex Aiken.

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.





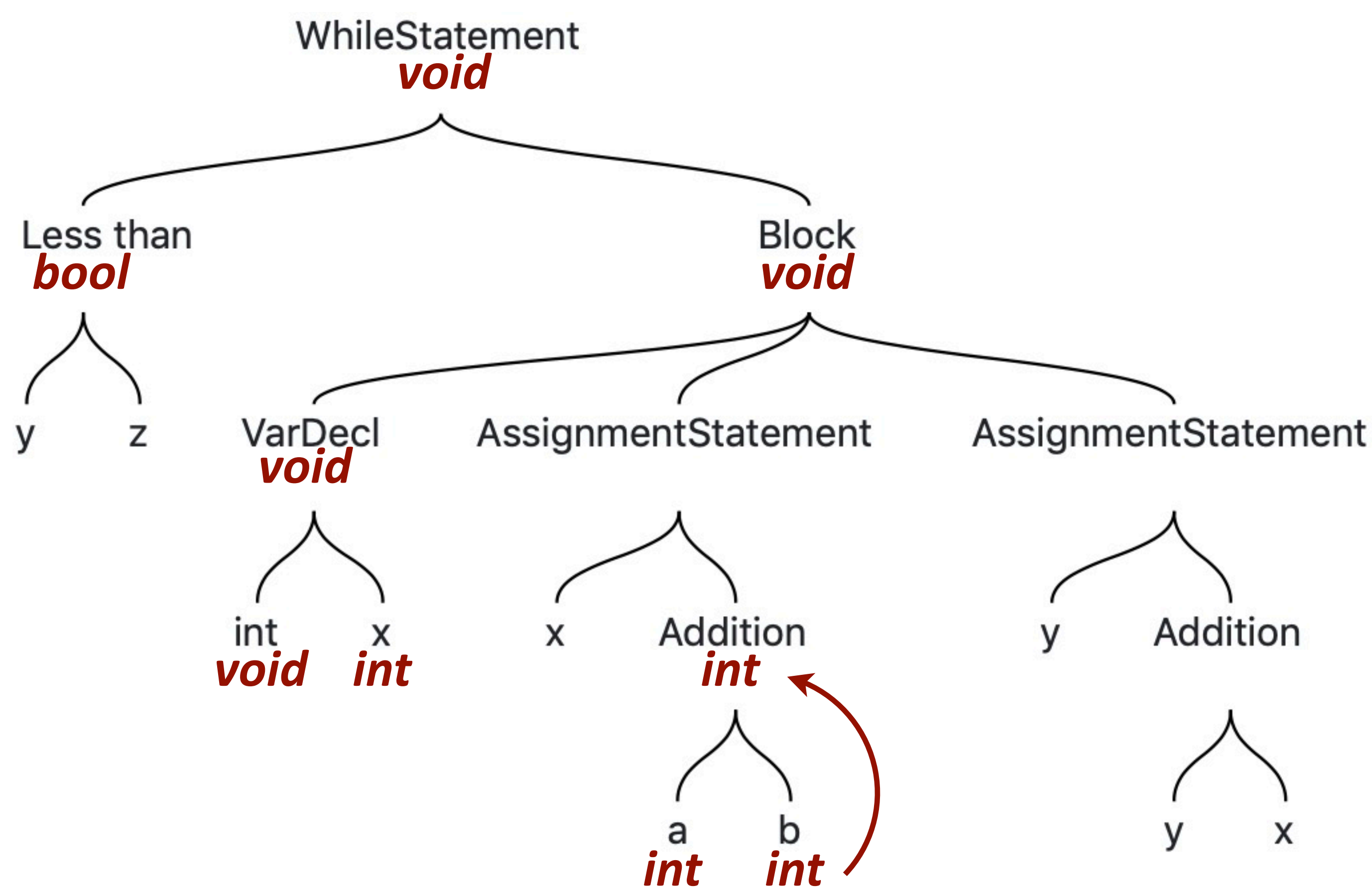
SA: Did we get the **meaning** (scope and **type**) right?

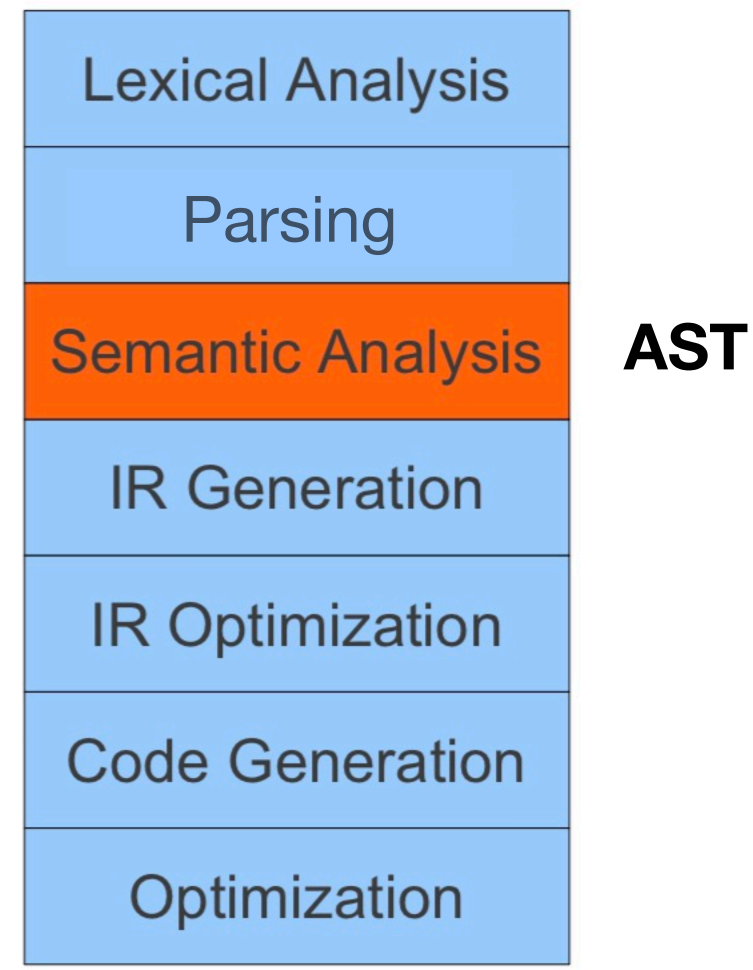
This is based on an example from the great Alex Aiken.

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.





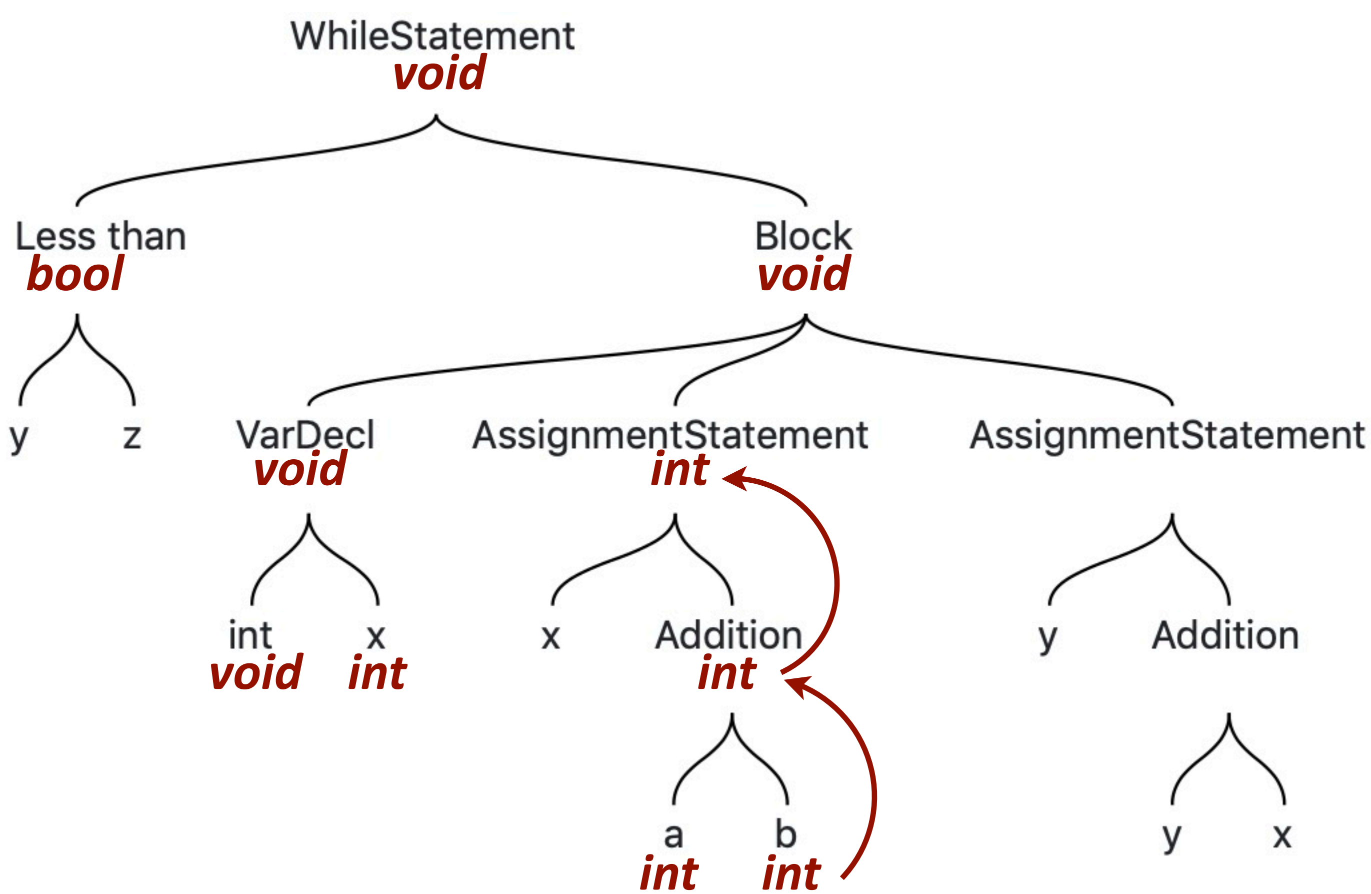
SA: Did we get the **meaning** (scope and **type**) right?

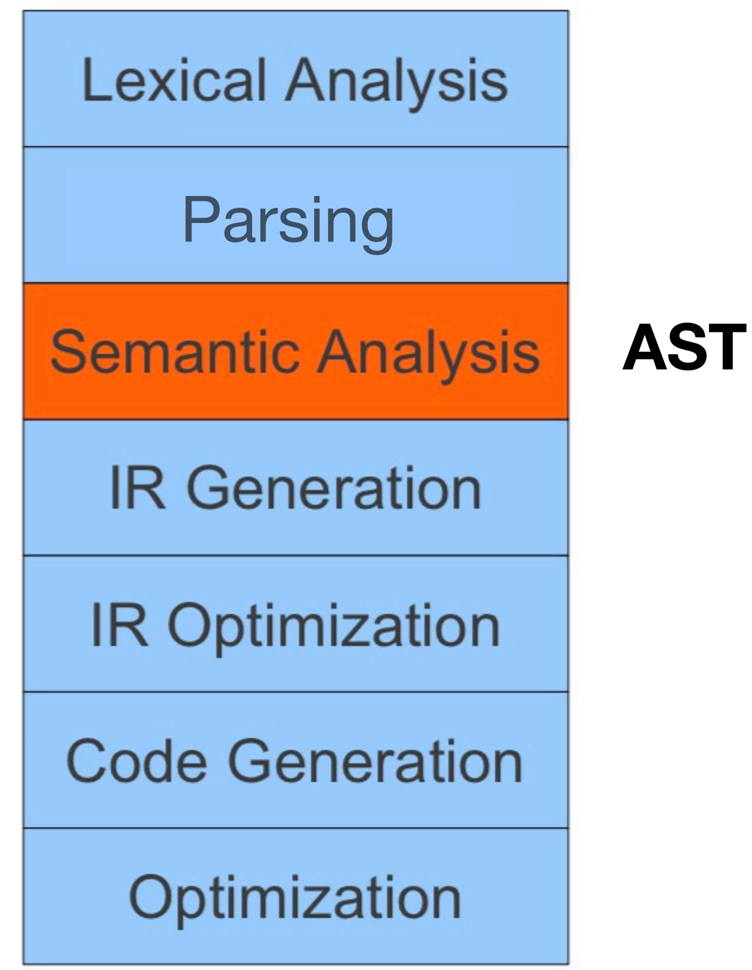
This is based on an example from the great Alex Aiken.

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.





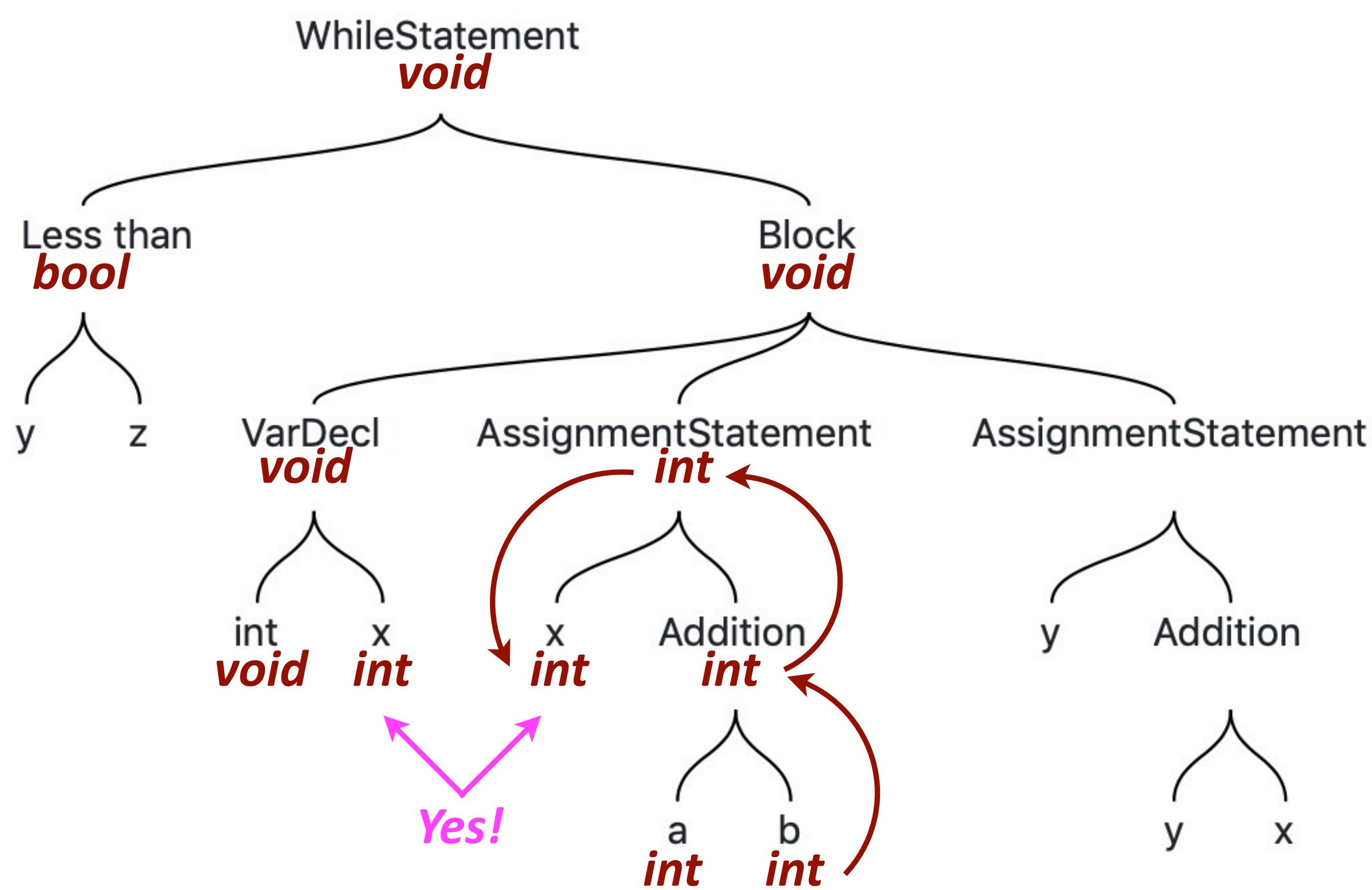
SA: Did we get the **meaning** (scope and **type**) right?

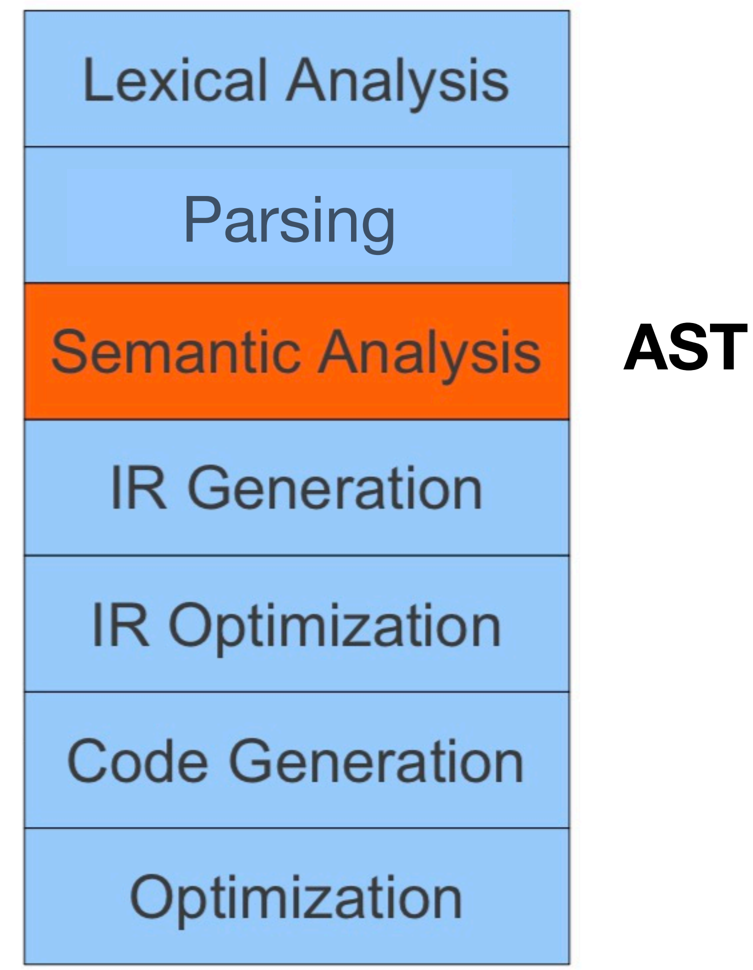
This is based on an example from the great Alex Aiken.

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.





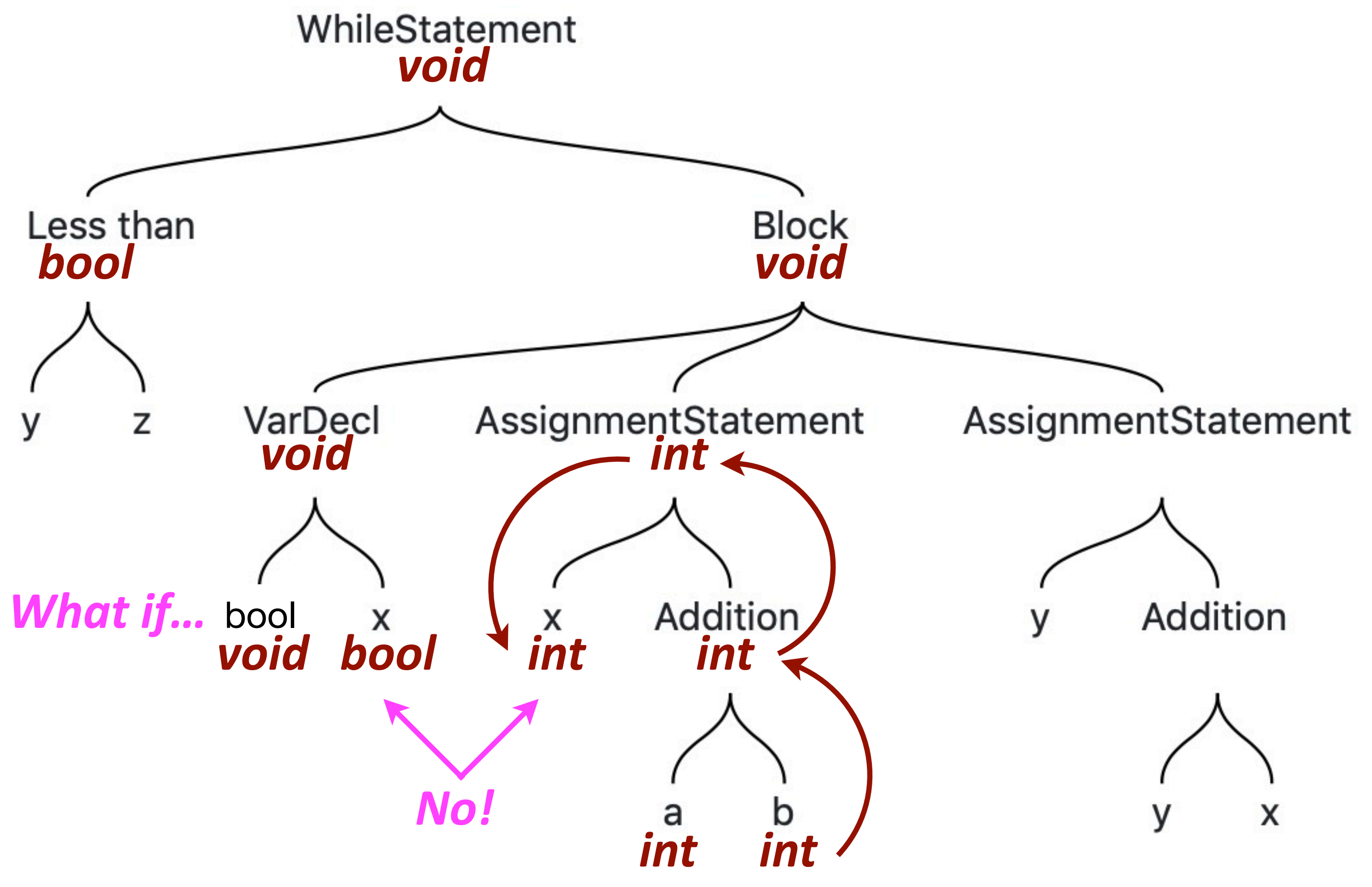
SA: Did we get the **meaning** (scope and **type**) right?

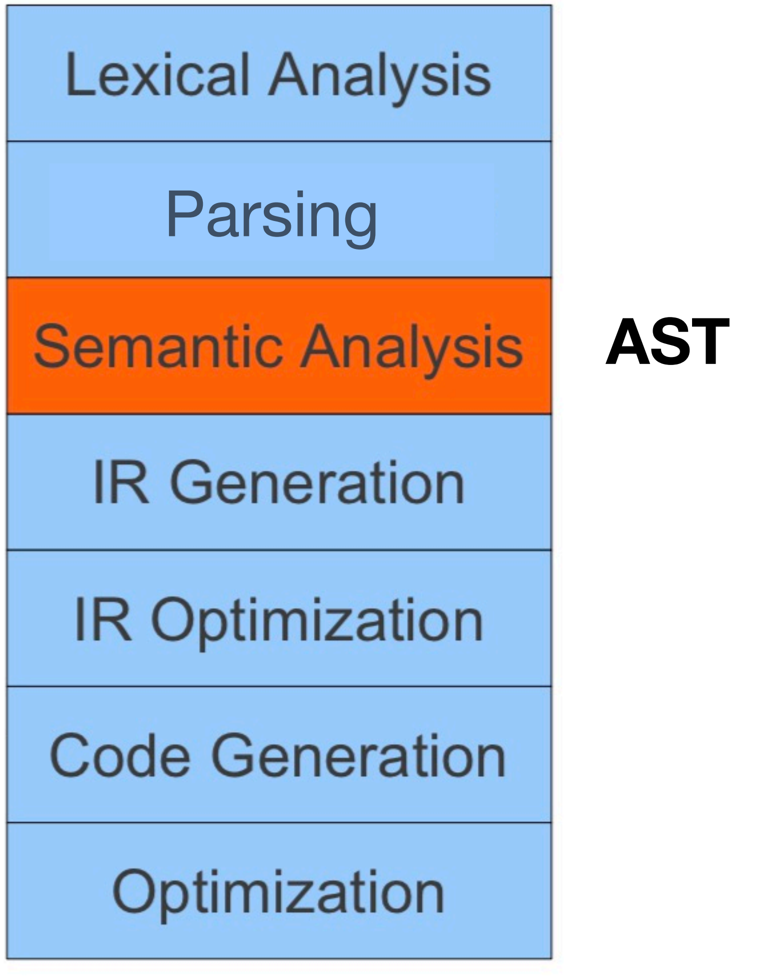
This is based on an example from the great Alex Aiken.

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.





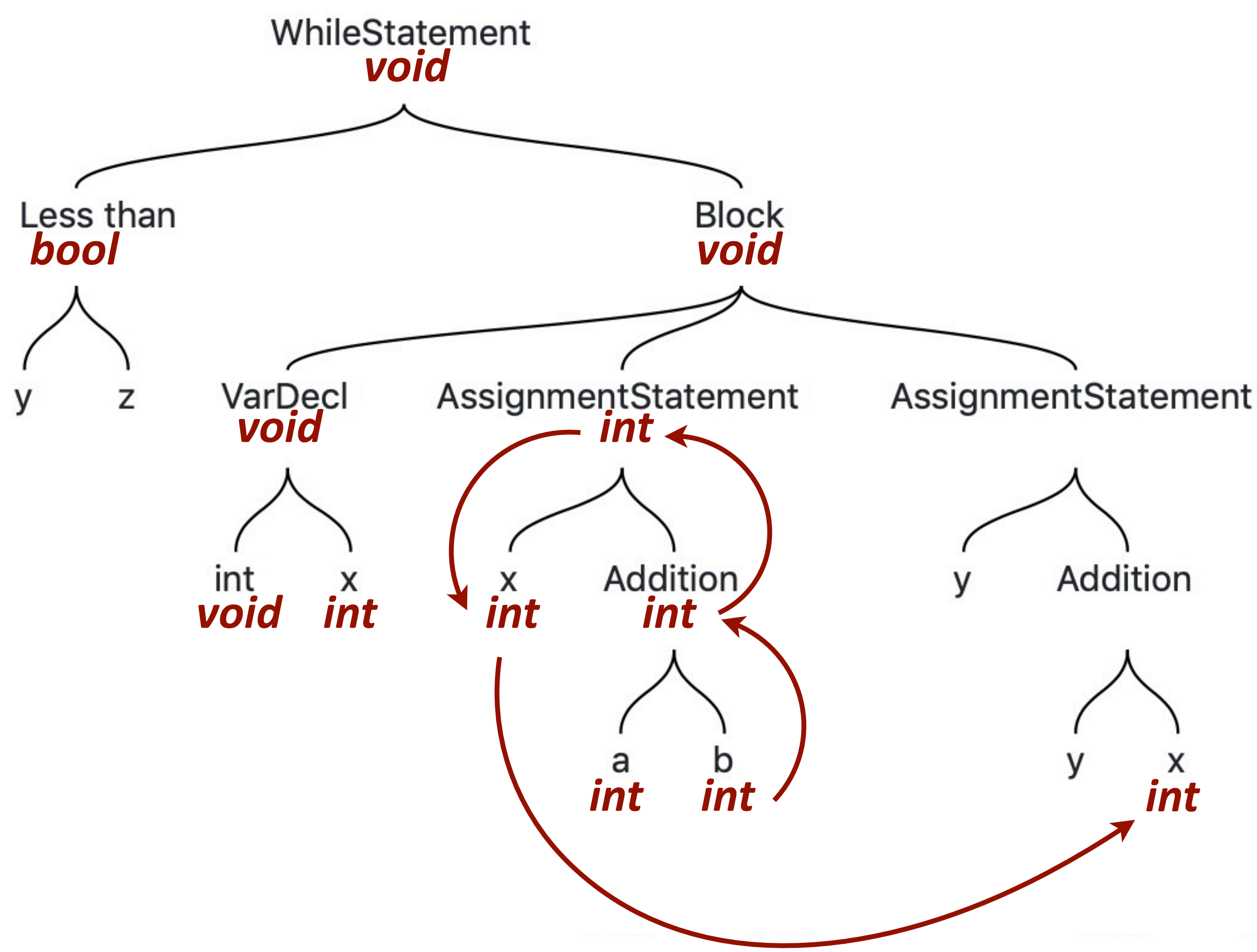
SA: Did we get the **meaning** (scope and **type**) right?

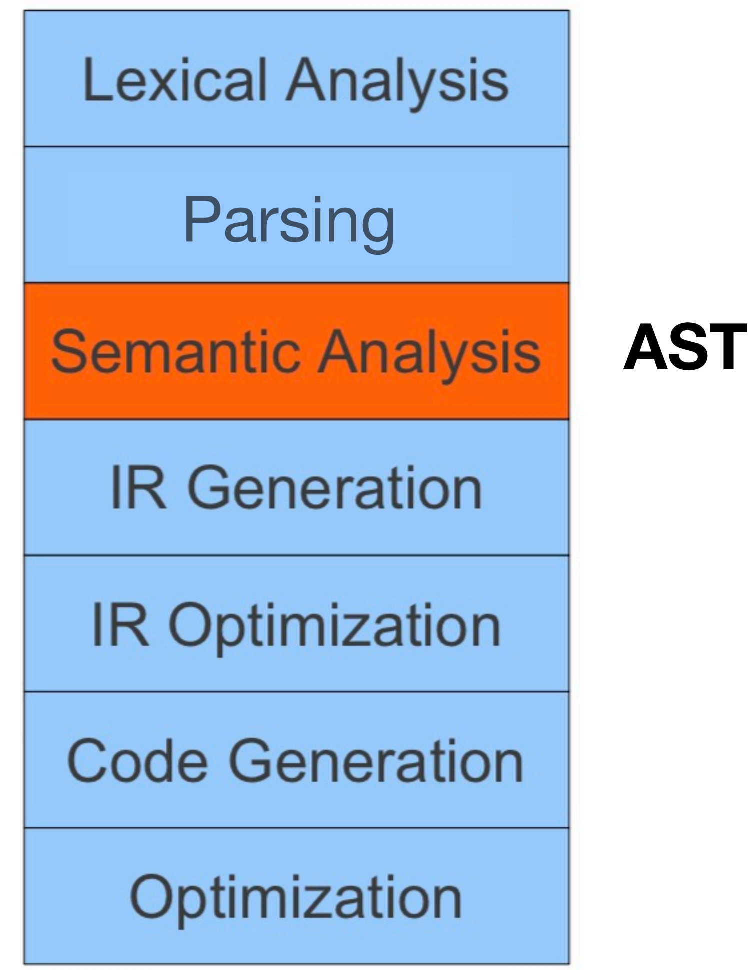
This is based on an example from the great Alex Aiken.

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.





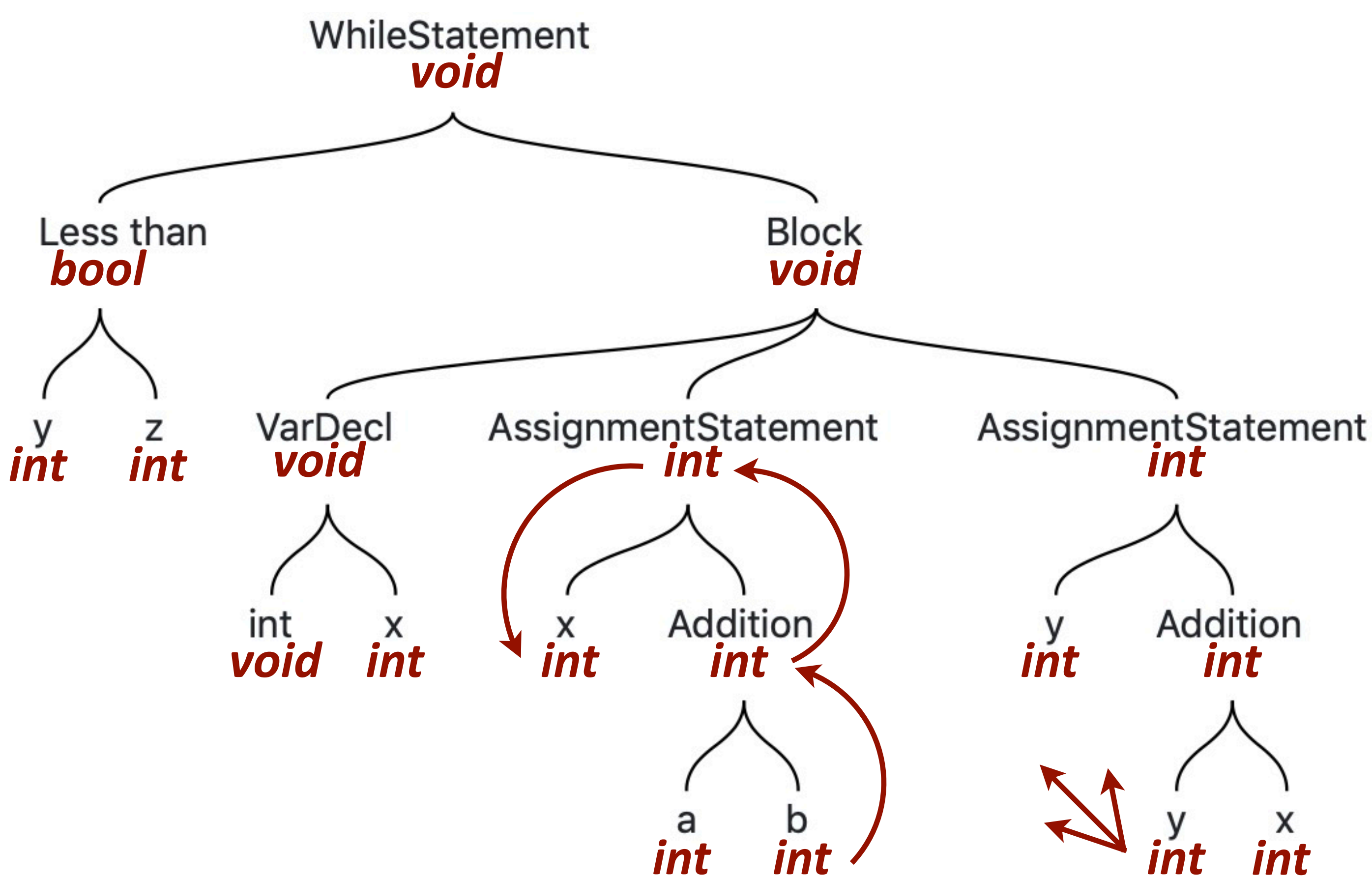
SA: Did we get the **meaning** (scope and **type**) right?

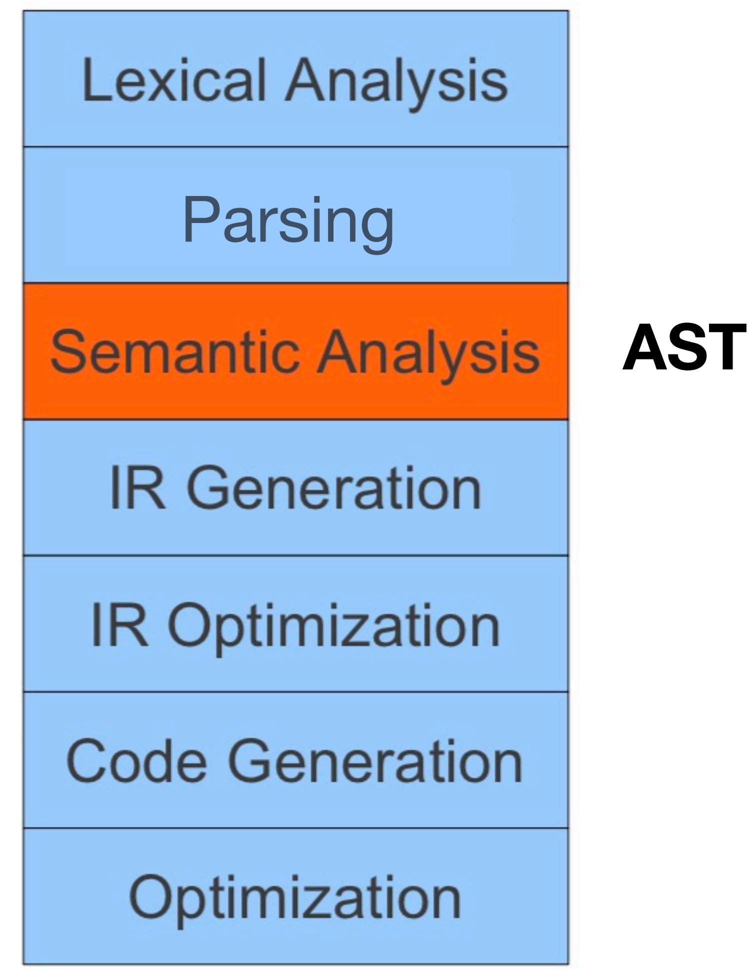
This is based on an example from the great Alex Aiken.

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.





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

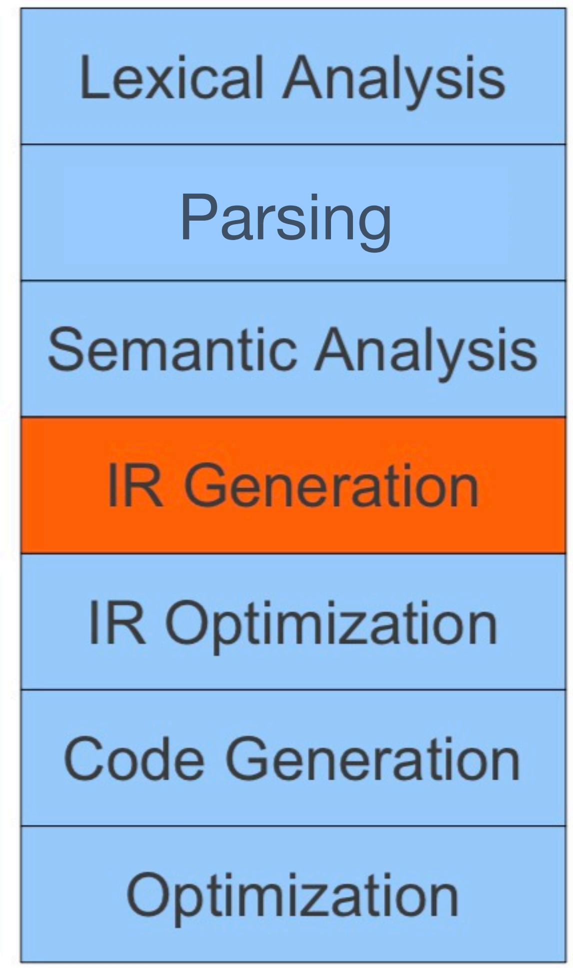
This is based on an example from the great Alex Aiken.

Example

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

} Assume these declarations.

```
Loop: x   = a + b
      y   = x + y
      _t1 = y < z
      if _t1 goto Loop
```



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

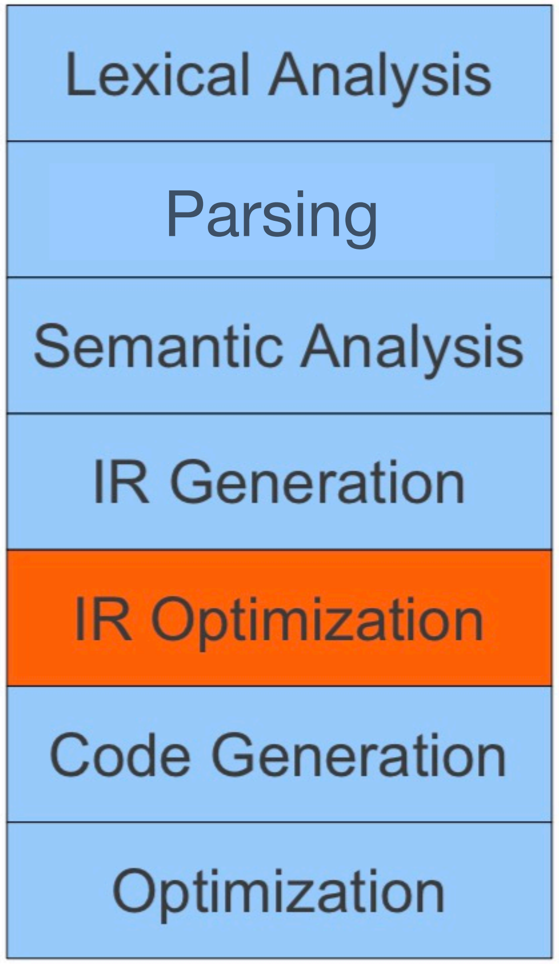
This is based on an example from the great Alex Aiken.

Example

```
const int a = 3
const int b = 4
```
} Assume these declarations.

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

```
        x   = a + b
Loop:   y   = x + y
        _t1 = y < z
        if _t1 goto Loop
```



IR: A “hoisting” optimization

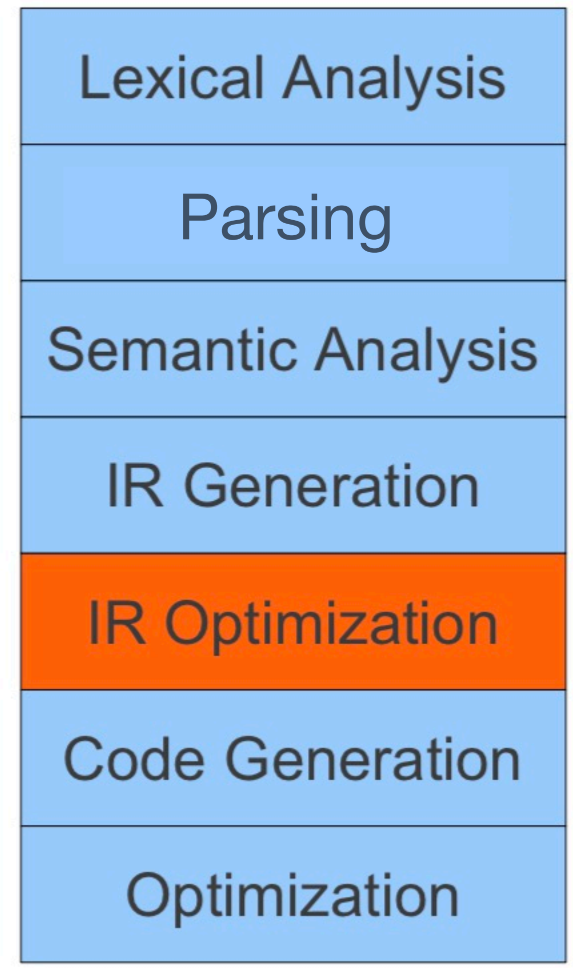
This is based on an example from the great Alex Aiken.

Example

```
const int a = 3
const int b = 4
```
} Assume these declarations.

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

x = ~~a + b~~ 7
Loop: y = x + y
_t1 = y < z
if _t1 goto Loop



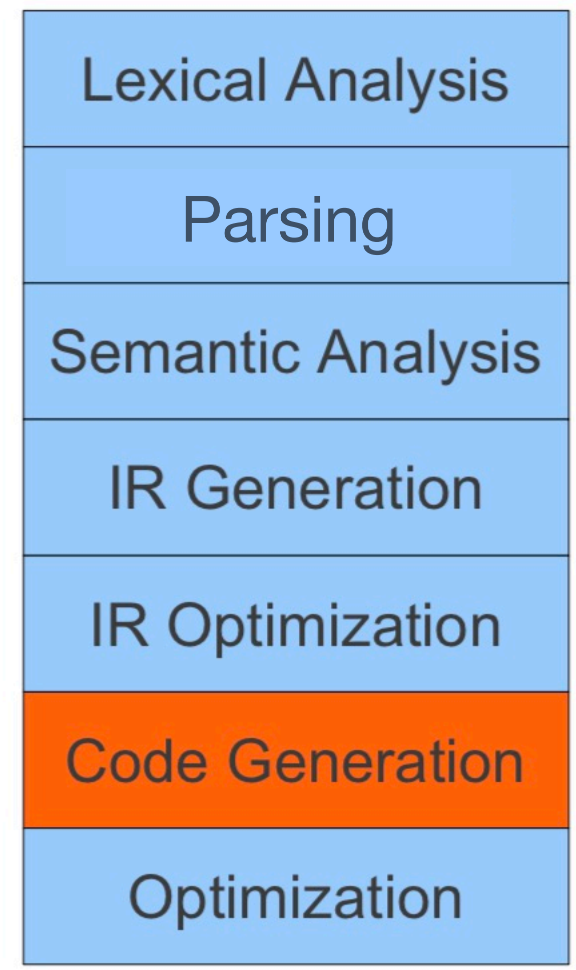
IR: A "constant folding" optimization

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. (Maybe RISC-V too.)

Example

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

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

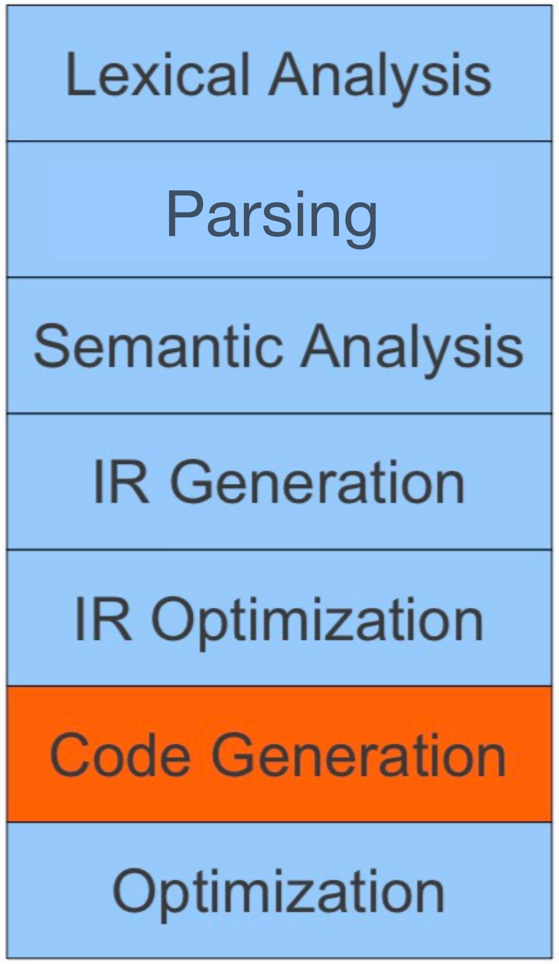
ML Optimization: replace *set* and *branch* with *branch less than*.

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

Example

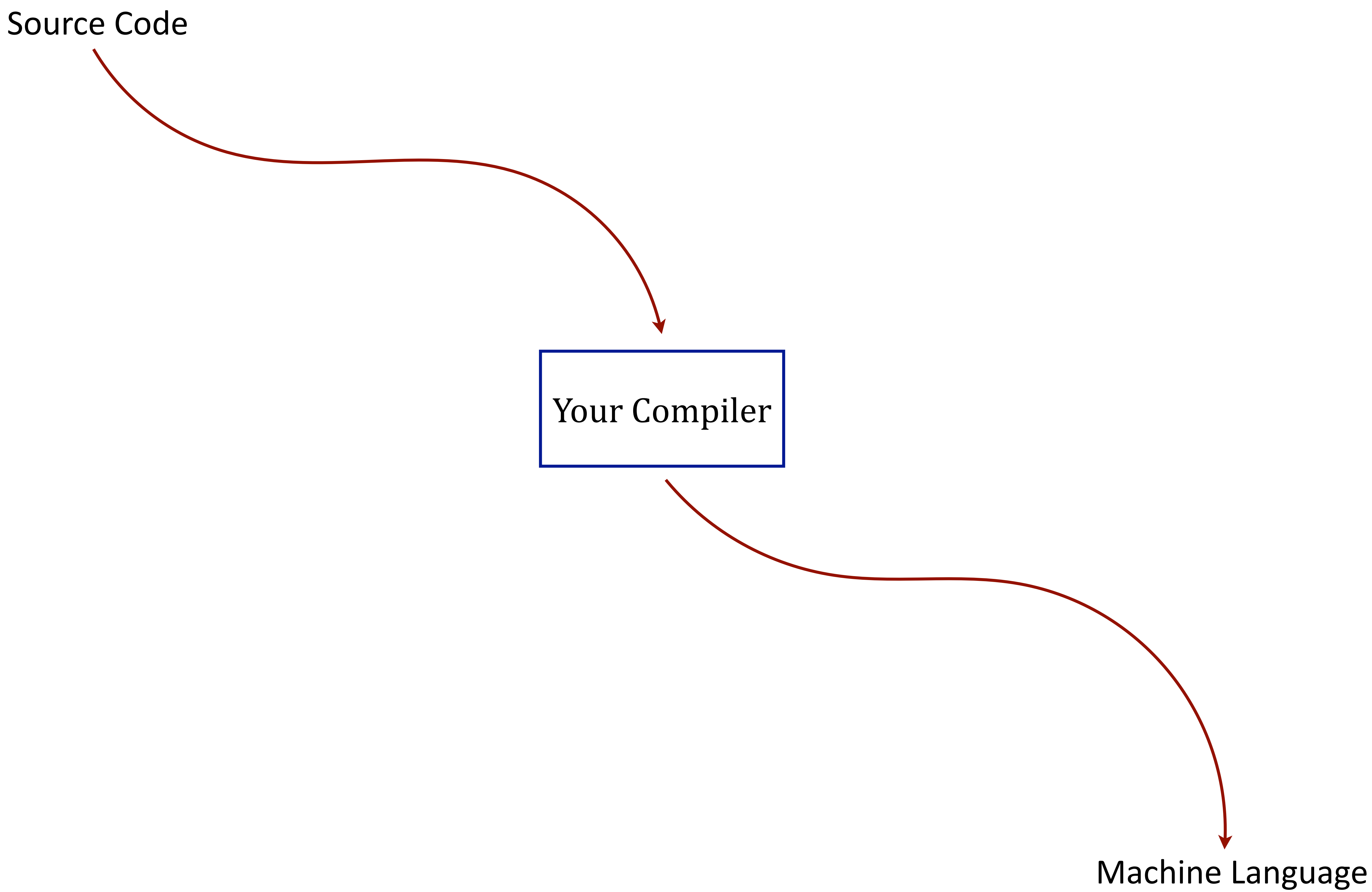
```
while (y < z) {
     int x = a + b;
     y += x;
}
```

```
// 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

Our Goal



I wonder . . .

Could we get an AI to write this for us?

This output[1] from ChatGPT3 on January 13, 2023 is not bad. It's not exactly what we're going to do in this class, but it's a good template and — best of all — a clean and well-structured design.

There are thousands of lines of detail to be written, but this is a decent start.

Write a program using typescript to simulate a compiler. This program should include 4 primary methods: lex, parse, analyze, generateCode. This program should accept a string of python code as its input and it should work through the four methods above to finally reach the codeGen method. CodeGen should convert the code to 6502a assembly language.

Here is an example program in TypeScript that simulates a compiler for Python code to 6502a assembly language:

Copy code

```
class Compiler {
  lex(code: string): any {
    // Lexical analysis code goes here
  }

  parse(tokens: any[]): any {
    // Parsing code goes here
  }

  analyze(ast: any): any {
    // Semantic analysis code goes here
  }

  generateCode(ir: any): string {
    // Code generation for 6502a assembly goes here
  }

  compile(code: string): string {
    const tokens = this.lex(code);
    const ast = this.parse(tokens);
    const ir = this.analyze(ast);
    return this.generateCode(ir);
  }
}

const compiler = new Compiler();
const assemblyCode = compiler.compile("print('Hello, World!')");
console.log(assemblyCode);
```

Note that the actual implementation of the lex, parse, analyze, and generateCode methods will depend on the specific requirements of the compiler and the 6502a assembly language. This is just a skeleton program to show the overall structure of the compiler.

1 This was generated by a recent (now graduated) Compilers student.

I wonder . . .

Could we get an AI to write this for us?

Here is an example implementation of the `lex` method for the provided Python code:

Copy code

```
lex(code: string): any {
  const tokens = [];
  let currentToken = "";
  for (let i = 0; i < code.length; i++) {
    const char = code[i];
    if (char === " " || char === "(" || char === ")" || char === "'") {
      if (currentToken.length > 0) {
        tokens.push(currentToken);
        currentToken = "";
      }
      if (char !== " ") {
        tokens.push(char);
      }
    } else {
      currentToken += char;
    }
  }
  if (currentToken.length > 0) {
    tokens.push(currentToken);
  }
  return tokens;
}
```

Here is an example of the `generateCode` method for converting the intermediate representation (IR) of the Python code to 6502a assembly language:

Copy code

```
generateCode(ir: any): string {
    let assemblyCode = "";
    for (let i = 0; i < ir.length; i++) {
        if (ir[i].op === "print") {
            assemblyCode += "LDA #" + ir[i].value + "\n";
            assemblyCode += "JSR $FFD2\n";
        }
    }
    return assemblyCode;
}
```

This example assumes that the IR is an array of objects, where each object has an `op` field that specifies the operation to be performed, and a `value` field that provides the value for the operation.

The function iterates over the IR, and for each print operation it generates the assembly code to load the value into the accumulator and call the JSR instruction to print it.

The assembly code generated by this example will look like this:

Copy code

```
LDA #'hello world'
JSR $FFD2
```

These *lex* and *codeGen* functions are neither complete nor accurate for our purposes, but they are both a "good start" towards that end.

I wonder . . .

~~Could we get an AI to write this for us~~~~?~~
Should we use an AI to help us write this?

Let's talk about it.

Midjourney
`> ChatGPT AI composing a C++ compiler software development retro-futuristic colorful bright space`

I wonder . . .

~~*Could we get an AI to write this for us*~~?
Should we use an AI to help us write this?

Let's talk about it.
You don't have to. Obviously.

Midjourney
```
> ChatGPT AI composing a C++ compiler
software development retro-futuristic
colorful bright space
```

I wonder . . .

~~*Could we get an AI to write this for us*?~~
Should we use an AI to help us write this?

Let's talk about it.
You don't have to. Obviously.
But if you do, here are some minimum requirements:

- As ever, any code that's not 100% yours and yours alone must be marked as such and documented with sources.
  - Whether you get help from ChatGPT3 online or Wilhelm in Marion Hall, you have to acknowledge it and cite it or it is unquestionably plagiarism.
  - Copying and pasting from the Hall of Fame or GitHub or some other online source is unquestionably plagiarism unless you cite it, so this is consistent.
- I'm interested in learning about how well (or not) AI-assisted software development worked for you.
  - If you make use of an AI in this project, you must document your experience with it for each project: what worked and what didn't, challenges, unexpected occurrences, etc. Include a reflection on the experience as well. You will soon be a professional in this field and I want to know your opinions on this.