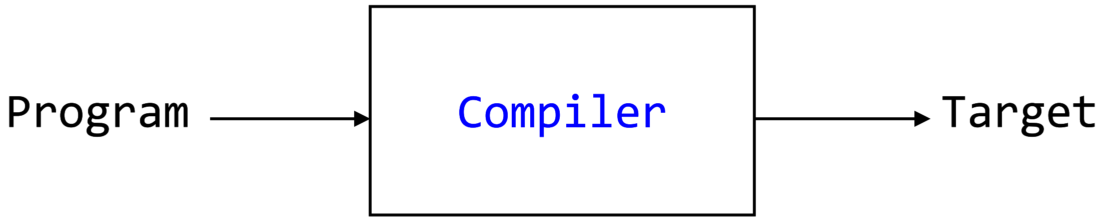
CS406: Compilers

Spring 2020

Week1: Overview, Structure of a compiler

Intro to Compilers

- Way to implement *programming languages*
    - Programming languages are notations for specifying computations to machines



Program

Compiler

Target

- *Target* can be an assembly code, executable, another source program etc.

What is a Compiler?

•Traditionally: Program that analyzes and **translates** from a high level language (e.g. C++) to low-level assembly language that can be executed by the hardware

```
int a, b;
a = 3;
if (a < 4) {
    b = 2;
} else {
    b = 3;
}
```

```
    var a
    var b
    mov 3 a
    mov 4 r1
    cmpi a r1
    jge l_e
    mov 2 b
    jmp l_d
l_e:mov 3 b
l_d:;done
```

Compilers are *translators*

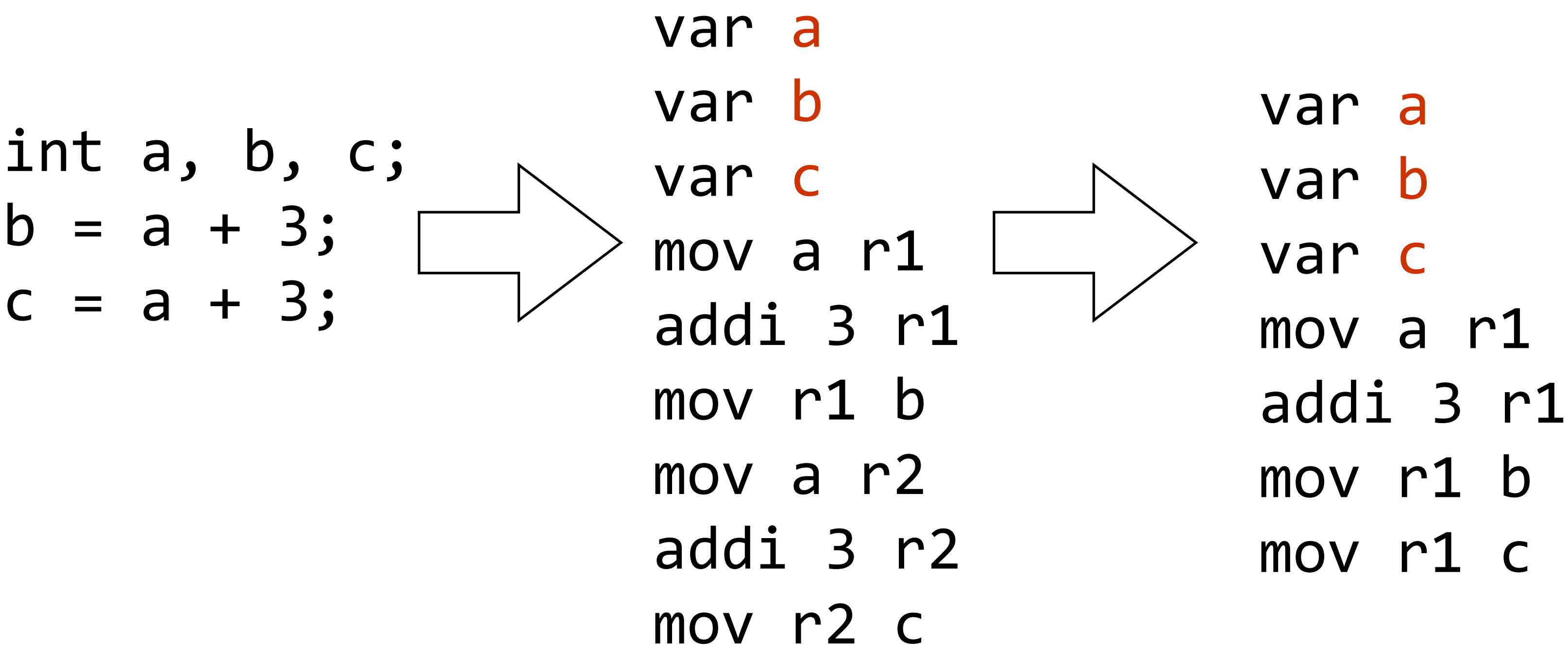
- Fortran
- C
- C++
- Java
- Text processing language
- HTML/XML
- Command & Scripting Languages
- Natural Language
- Domain Specific Language

translate

- Machine code
- Virtual machine code
- Transformed source code
- Augmented source code
- Low-level commands
- Semantic components
- Another language

Compilers are *optimizers*

- Can perform optimizations to make a program more efficient



```
int a, b, c;
b = a + 3;
c = a + 3;
```

```
var a
var b
var c
mov a r1
addi 3 r1
mov r1 b
mov a r2
addi 3 r2
mov r2 c
```

```
var a
var b
var c
mov a r1
addi 3 r1
mov r1 b
mov r1 c
```

Why do we need compilers?

- Compilers provide *portability*
- Old days: whenever a new machine was built, programs had to be rewritten to support new instruction sets
- IBM System/360 (1964): Common Instruction Set Architecture (ISA) --- programs could be run on any machine which supported ISA
  - Common ISA is a huge deal (note continued existence of x86)
- But still a problem: when new ISA is introduced (EPIC) or new extensions added (x86-64), programs would have to be rewritten
- Compilers bridge this gap: write new compiler for an ISA, and then simply recompile programs!

Why do we need compilers?

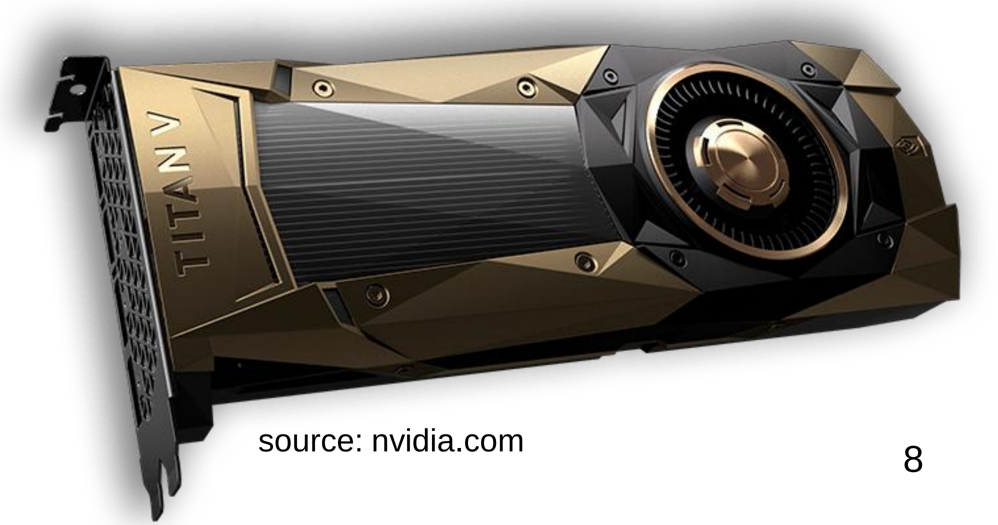
- Compilers enable high-performance and productivity
- Old: programmers wrote in assembly language, architectures were simple (no pipelines, caches, etc.)
  - Close match between programs and machines --- easier to achieve performance
- New: programmers write in high level languages (Ruby, Python), architectures are complex (superscalar, out-of-order execution, multicore)
- Compilers are needed to bridge this *semantic gap*
  - Compilers let programmers write in high level languages and still get good performance on complex architectures

Semantic Gap

- Python code that actually runs on GPU

```
import pycuda
import pycuda.autoinit from pycuda.tools import
make_default_context
c = make_d
d = c.get_
……
```

Impossible without Compilers

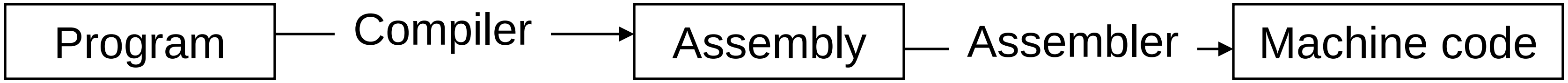


source: nvidia.com

Some common compiler types

- High level language ⟹ assembly language (e.g. gcc)
- High level language ⟹ machine independent bytecode (e.g. javac)
- Bytecode ⟹ native machine code (e.g. java’s JIT compiler)
- High level language ⟹ High level language (e.g. domain specific languages, many research languages)

HLL to Assembly

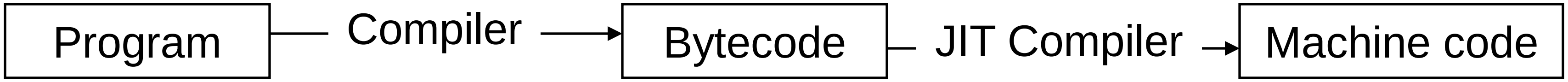


- Compiler converts program to assembly
- Assembler is machine-specific translator which converts assembly to machine code

```
add $7 $8 $9 ($7 = $8 + $9 ) => 000000 00111 01000 01001 00000 100000
```

- Conversion is usually one-to-one with some exceptions
  - Program locations
  - Variable names

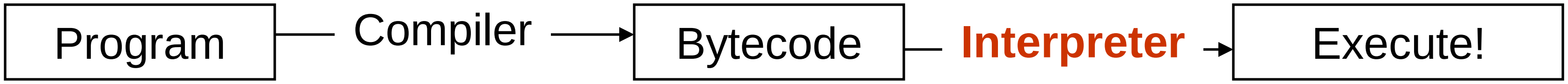
HLL to Bytecode to Assembly



Program → Compiler → Bytecode → JIT Compiler → Machine code

- Compiler converts program into machine independent bytecode
  - e.g. javac generates Java bytecode, C# compiler generates CIL
- Just-in-time compiler compiles code *while program executes* to produce machine code
  - Is this better or worse than a compiler which generates machine code directly from the program?

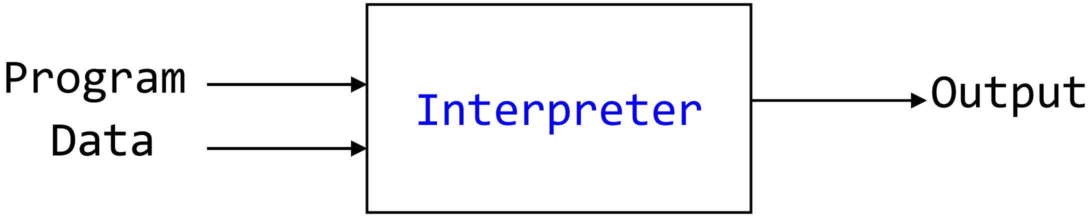
HLL to Bytecode

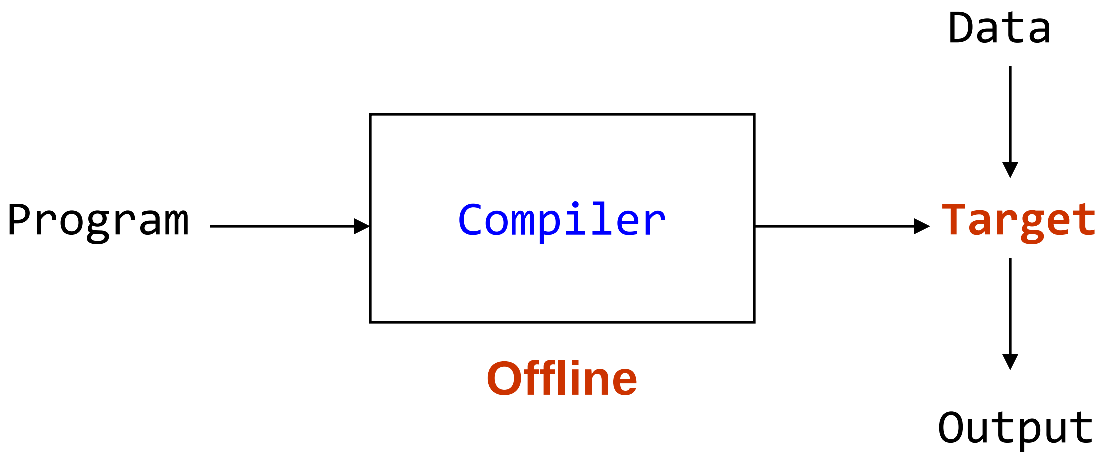


- Compiler converts program into machine independent bytecode
    - e.g. javac generates Java bytecode, C# compiler generates CIL
- Interpreter then executes bytecode “on-the-fly”
- Bytecode instructions are “executed” by invoking methods of the interpreter, rather than directly executing on the machine
- Aside: what are the pros and cons of this approach?

Quick Detour: Interpreters

- Alternate way to implement programming languages





Data

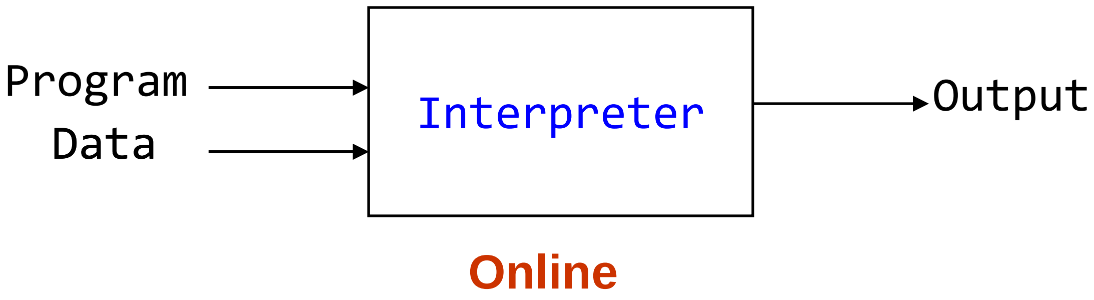
Program

Compiler

Target

Offline

Output



Program

Data

Interpreter

Output

Online

these are the two types of language processing systems

History

- 1954: IBM 704
  - Huge success
  - Could do complex math
  - Software cost > Hardware cost

Source: IBM Italy, https://commons.wikimedia.org/w/index.php?curid=48929471

How can we improve the efficiency of creating software?

- 1953: Speedcoding
  - *High-level programming language* by John Backus
  - Early form of *interpreters*
  - Greatly reduced programming effort

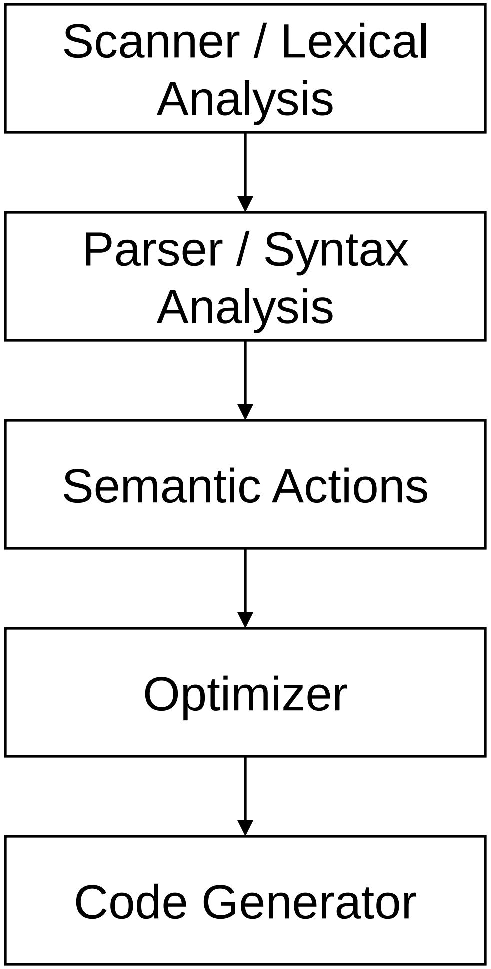
  - About 10x-20x slower
  - Consumed lot of memory (~300 bytes = about 30% RAM)

Fortran I

- 1957: Fortran released
  - Building the compiler took 3 years
  - Very successful: by 1958, 50% of all software created was written in Fortran
- Influenced the design of:
  - high-level programming languages e.g. BASIC
  - practical compilers

Today's compilers still preserve the structure of Fortran I

Structure of a Compiler



Scanner / Lexical Analysis

Parser / Syntax Analysis

Semantic Actions

Optimizer

Code Generator

Scanner

- A compiler starts by seeing only program text

```
if ( a < 4) {
    b = 5
}
```

- Analogy: Humans processing English text

```
Rama is a neighbor.
```

Scanner

- A compiler starts by seeing only program text

```
‘i’ ‘f’ ‘ ’ ‘(’ ‘a’ ‘<’ ‘4’ ‘)’
 ‘ ’ ‘{’ ‘\n’ ‘\t’ ‘b’ ‘=’ ‘5’
        ‘\n’ ‘}’
```

Scanner

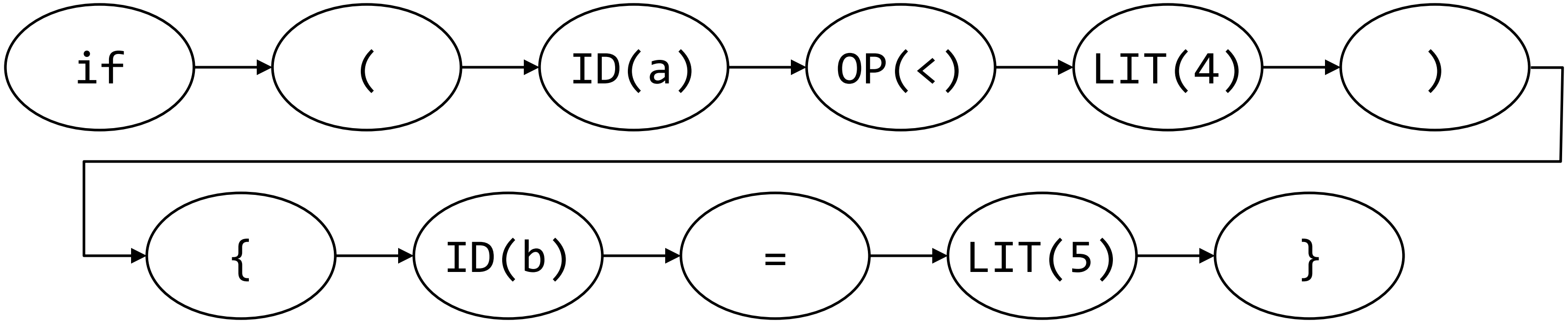
- A compiler starts by seeing only program text
- Scanner converts program text into string of *tokens*

```
‘i’ ‘f’ ‘ ’ ‘(’ ‘a’ ‘<’ ‘4’ ‘)’
 ‘ ’ ‘{’ ‘\n’ ‘\t’ ‘b’ ‘=’ ‘5’
          ‘\n’ ‘}’
```

- Analogy: Humans processing English text
  - recognize words
    - Rama, is, a, neighbor
    - Additional details such as punctuations, capitalizations, blankspaces etc.

Scanner

- A compiler starts by seeing only program text
- Scanner converts program text into string of *tokens*



- But we still don’t know what the *syntactic structure* of the program is

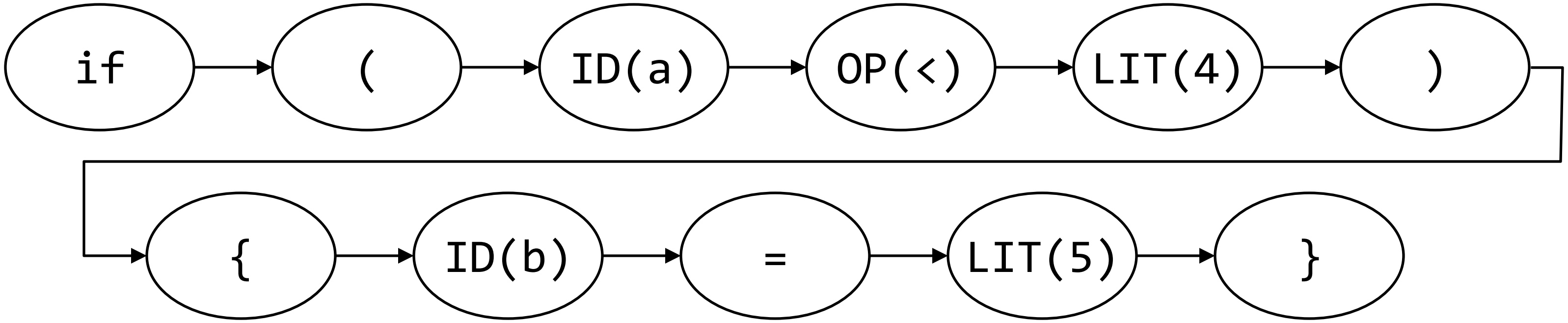
Exercise

Convert the following program text into tokens:

```
pos = initPos + speed * 60
```

Parser

- Converts a string of tokens into *parse tree* or *abstract syntax tree*
- Captures syntactic structure of the code (i.e. “this is an `if` statement, with a `then`-block”

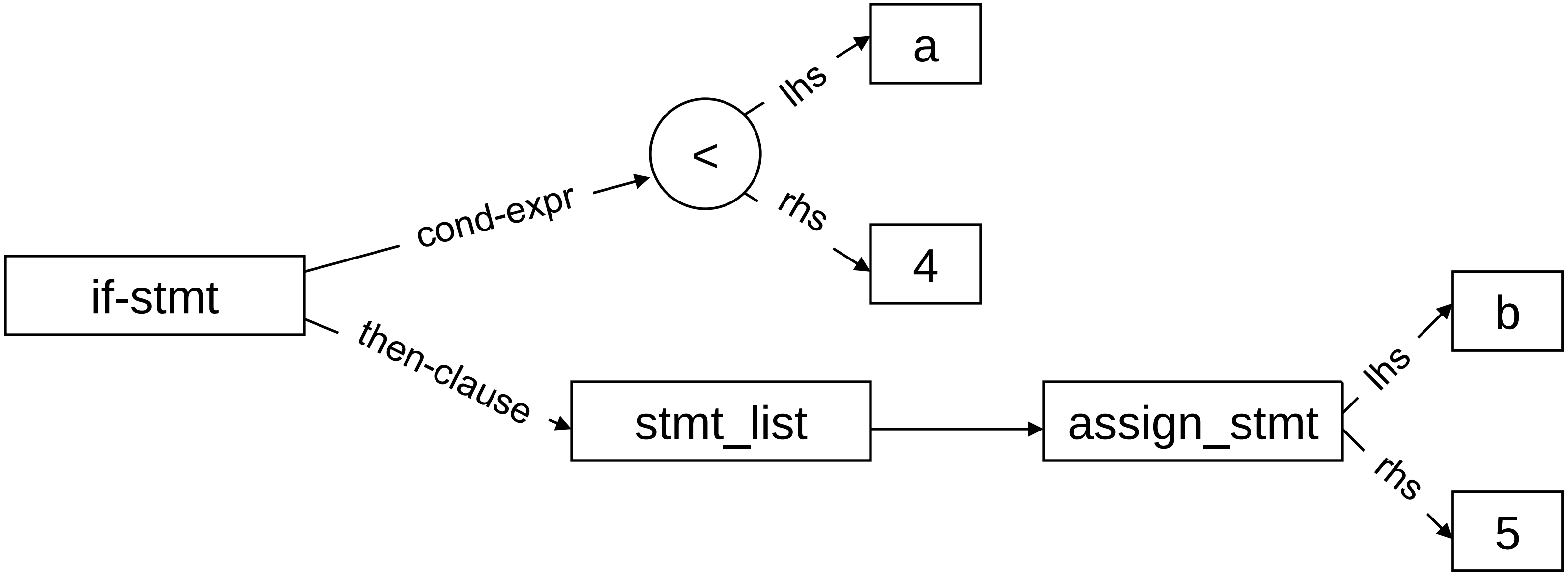


- Analogy: understand the English sentence structure

```
Rama is a good neighbor
```

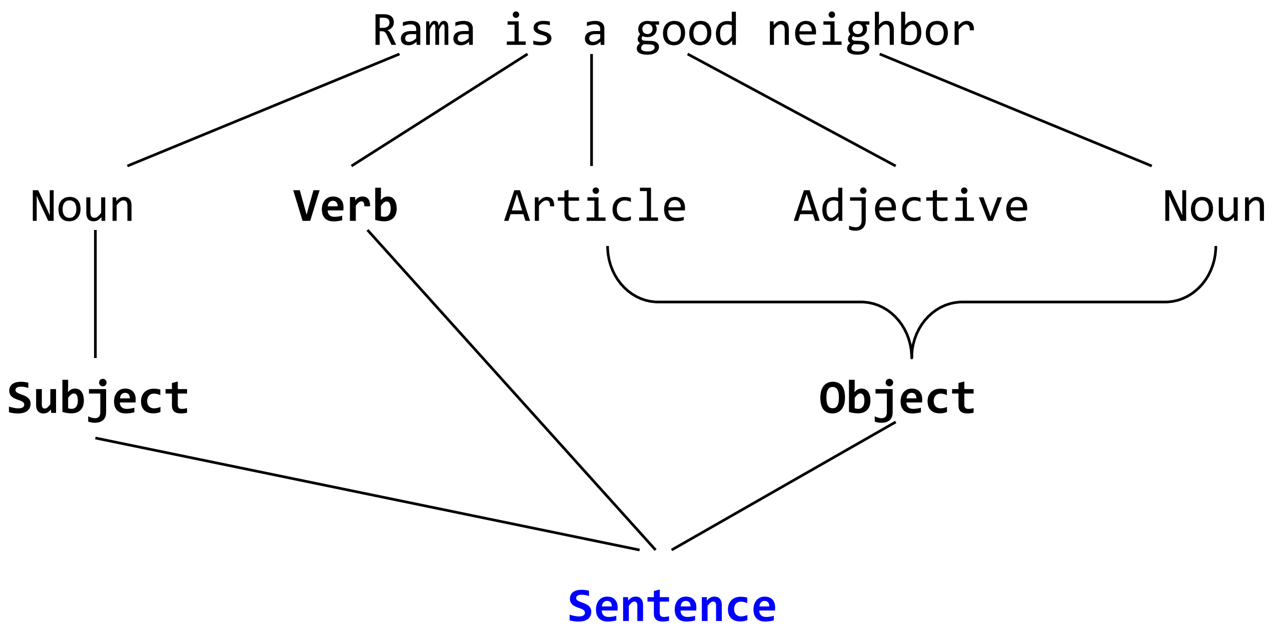
Parser

- Converts a string of tokens into *parse tree* or *abstract syntax tree*
- Captures syntactic structure of the code (i.e. “this is an `if` statement, with a `then`-block”



Parser - Analogy

- Diagramming English sentences



Exercise

Draw the syntax tree for the following program stmt:

```
pos = initPos + speed * 60
```

Semantic Actions

- Interpret the *semantics* of syntactic constructs

- Refer to actions taken by the compiler based on the *semantics* of program statements.

- Up until now, we have looked at syntax of a program
  - *what is the difference?*

Syntax vs. Semantics

- Syntax: “grammatical” structure of language
  - What symbols, in what order, is a legal part of the language?
    - But something that is syntactically correct may mean nothing!
    - “colorless green ideas sleep furiously”

- Semantics: meaning of language
  - What does a particular set of symbols, in a particular order *mean?*
    - What does it mean to be an if statement?
    - “evaluate the conditional, if the conditional is true, execute the then clause, otherwise execute the else clause”

Semantic Actions - What

- What actions are taken by compiler based on the semantics of program statements ?
  - Examples:

    - bind variables to their scopes

    - check for type inconsistencies
- Analogy:

  - Raj said Raj has a big heart

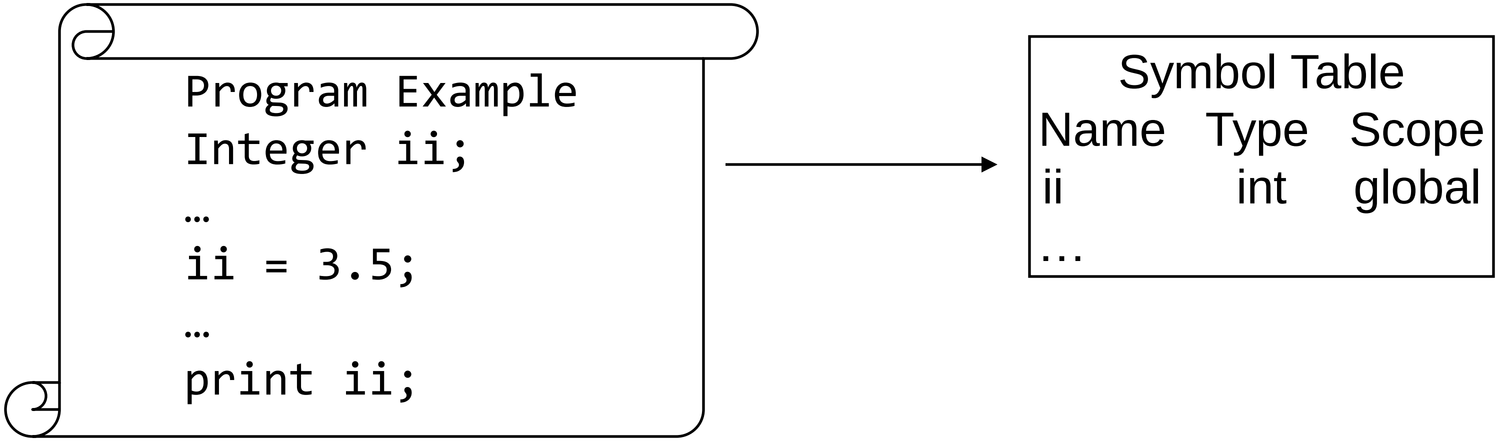
  - Raj left her home in the evening

Semantic Actions - How

- What actions are taken by compiler based on the semantics of program statements ?
  - Building a *symbol table*
  - Generating *intermediate representations*

Symbol Tables

- A list of every declaration in the program, along with other information
  - Variable declarations: types, scope
  - Function declarations: return types, # and type of arguments



```
Program Example
Integer ii;
…
ii = 3.5;
…
print ii;
```

Symbol Table

Name	Type	Scope
ii	int	global
…		

Intermediate Representation

- Also called *IR*
- A (relatively) low level representation of the program
  - But not machine-specific!
- One example: *three address code*

```
        bge a, 4, done
        mov 5, b
    done: //done!
```

- Each instruction can take at most three operands (variables, literals, or labels)
  - Note: no registers!

Exercise

Explain the semantics of the following program stmt:

```
pos = initPos + speed * 60
```

A Note on Semantics

- How do you define semantics?
  - **Static semantics:** properties of programs
    - All variables must have type
    - Expressions must use consistent types
    - Can define using *attribute grammars*
  - **Execution semantics:** how does a program execute?
    - Defined through *operational* or *denotational* semantics
    - Beyond the scope of this course!
  - For many languages, “the compiler is the specification”

Optimizer

- Transforms code to make it more efficient
- Different kinds, operating at different levels
  - High-level optimizations
    - Loop interchange, parallelization
    - Operates at level of AST, or even source code
  - Scalar optimizations
    - Dead code elimination, common sub-expression elimination
    - Operates on IR
  - Local optimizations
    - Strength reduction, constant folding
    - Operates on small sequences of instructions

Optimizer - Analogy

Analogy: reducing word usage

Dejavu

Sunny felt a sense of ~~having experienced it before~~ when his bike broke down.

Exercise: *is this rule correct?*

`X = Y * 0` is the same as `X = 0`

Code Generation

- Generate assembly from intermediate representation
  - Select which instruction to use
  - Select which register to use
  - Schedule instructions

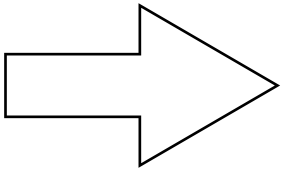
```
bge a, 4 done
mov 5, b
done: //done
```

```
ld  a, r1
mov 4, r2
cmp r1, r2
bge done
mov 5, r3
st r3, b
done:
```

Code Generation

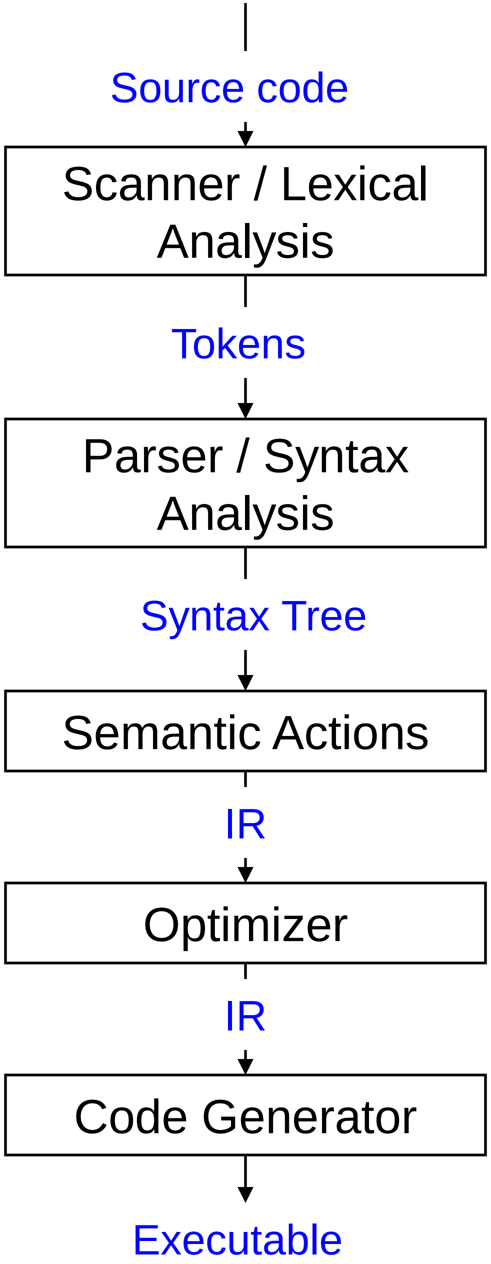
- Generate assembly from intermediate representation
  - Select which instruction to use
  - Select which register to use
  - Schedule instructions

```
bge a, 4 done
mov 5, b
done: //done
```



```
mov 4, r1
ld  a, r2
cmp r1, r2
blt done
mov 5, r1
st r1, b
done:
```

Structure of a Compiler



Source code

Scanner / Lexical Analysis

Tokens

Parser / Syntax Analysis

Syntax Tree

Semantic Actions

IR

Optimizer

IR

Code Generator

Executable

Use *regular expressions* to define tokens. Can then use scanner generators such as lex or flex.

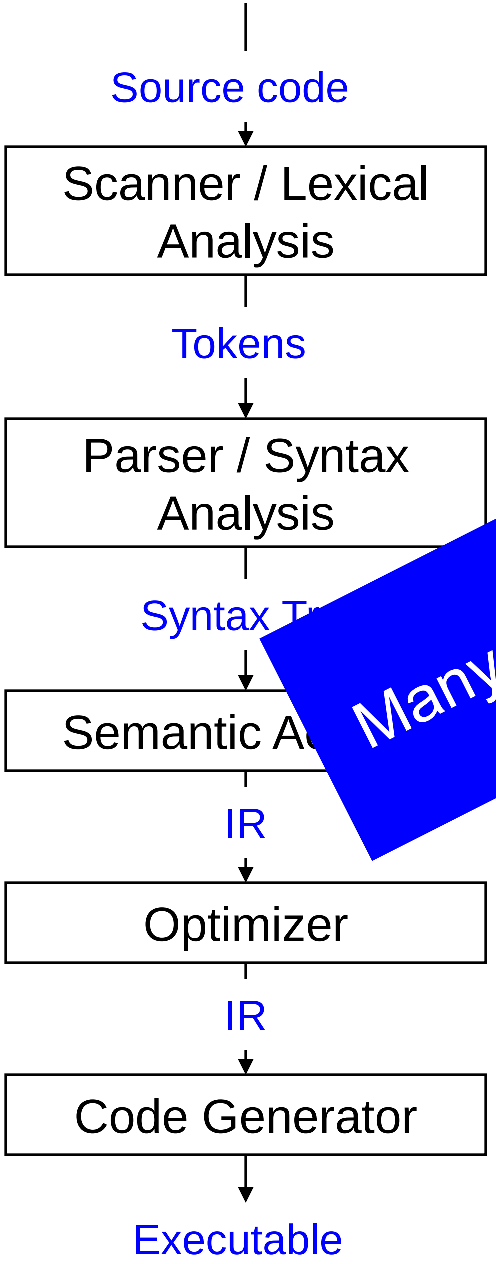
Define language using *context free grammars.* Can then use parser generators such as yacc or bison.

Semantic routines done by hand. But can be formalized.

Written manually. Automation is an active research area (e.g. dataflow analysis frameworks)

Written manually.

Structure of a Compiler



Source code

Scanner / Lexical Analysis

Tokens

Parser / Syntax Analysis

Syntax Tr

Semantic A

IR

Optimizer

IR

Code Generator

Executable

Use *regular expressions* to ens. Can then use
scanner gen or flex.

D *ntext free grammars.* Can then
erators such as yacc or bison.

antic routines done by hand. But can be formalized.

Many of these passes can be combined

Written manually. Automation is an active research area
(e.g. dataflow analysis frameworks)

Written manually.

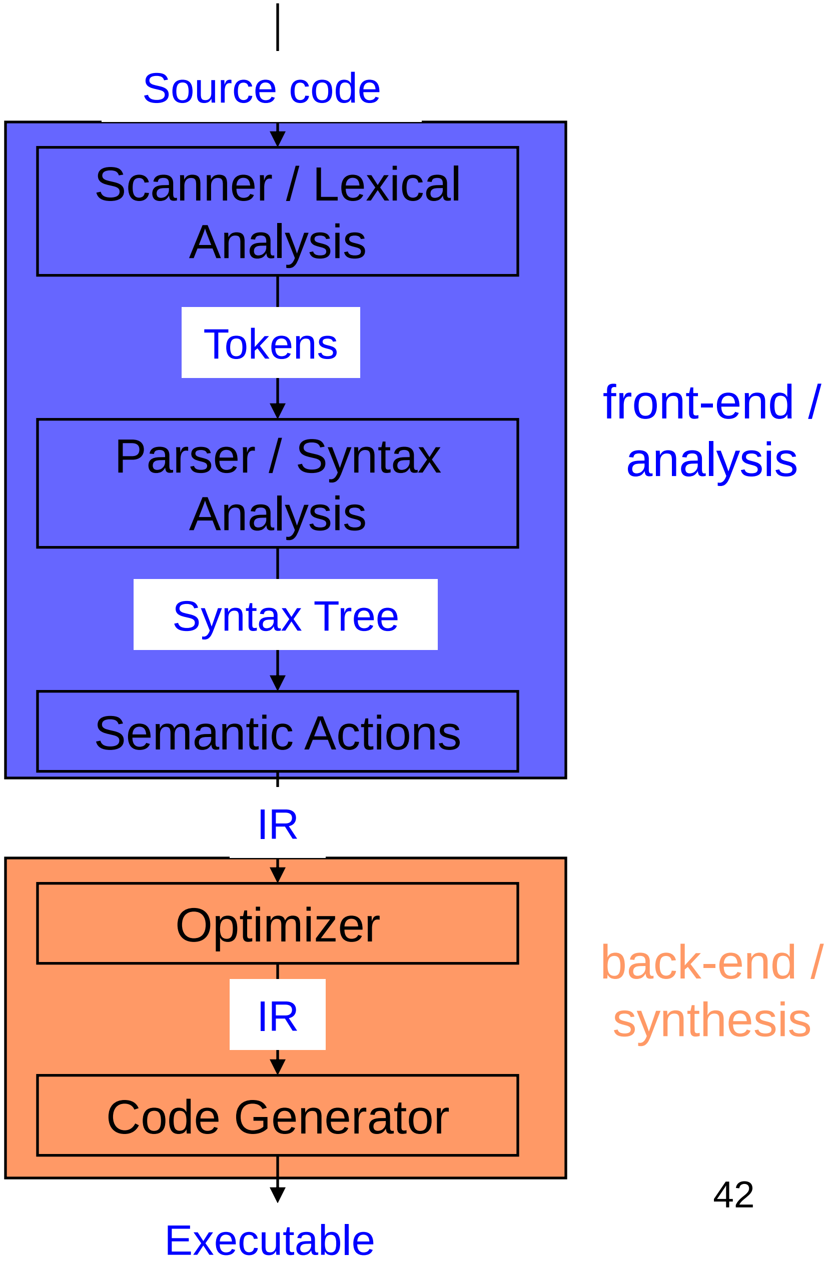
Front-end vs. Back-end

- Scanner + Parser + Semantic actions + (high level) optimizations called the *front-end* of a compiler

- IR level optimizations and code generation (instruction selection, scheduling, register allocation) called the *back-end* of a compiler

- Can build multiple front-ends for a particular back-end
  - e.g. gcc or g++ or many front-ends which generate CIL

- Can build multiple back-ends for a particular front-end
  - gcc allows targeting different architectures



Programming Language Design Considerations

- Why are there so many programming languages?
- Why are there new languages?
- What is a good programming language?

- Compiler and language designs influence each other
  - Higher level languages are harder to compile
    - More work to bridge the gap between language and assembly
  - Flexible languages are often harder to compile
    - Dynamic typing (Ruby, Python) makes a language very flexible, but it is hard for a compiler to catch errors (in fact, many simply won’t)
  - Influenced by architectures
    - RISC vs. CISC