

Language Translators

Ronghui Gu

Spring 2022

Columbia University

* Course website: <https://verigu.github.io/4115Spring2022/>

** These slides are borrowed from Prof. Edwards.

TA Mailing List

For any general questions related to assignments and projects, please send emails to the following TA mailing list using your **Columbia email address**:

4115Spring2019TA@lists.cs.columbia.edu

What is a Translator?

A programming language is a notation that a person and a computer can both understand.

- It allows you to express what is the **task** to compute
- It allows a computer to **execute** the computation task

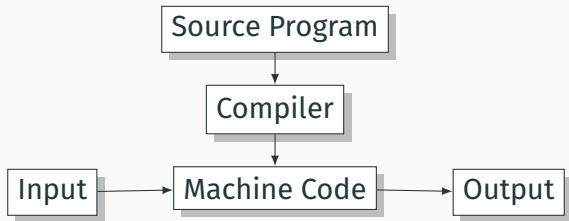
What is a Translator?

A programming language is a notation that a person and a computer can both understand.

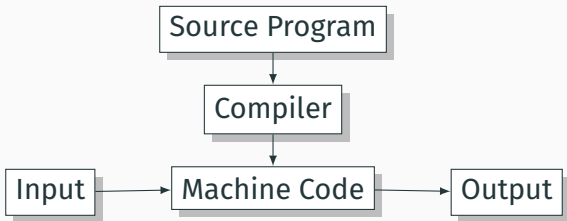
- It allows you to express what is the **task** to compute
- It allows a computer to **execute** the computation task

A translator translates what you express to what a computer can execute.

Compiler

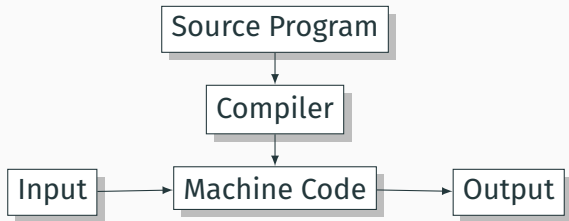


Compiler



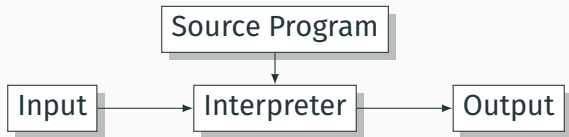
- **Pros:** translation is done once and for all; optimize code and map identifiers at compile time.

Compiler

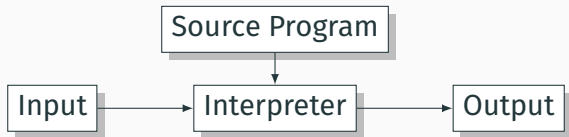


- **Pros:** translation is done once and for all; optimize code and map identifiers at compile time.
- **Cons:** long compilation time; hard to port.

Interpreter

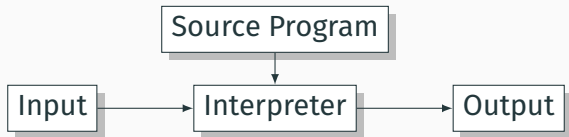


Interpreter



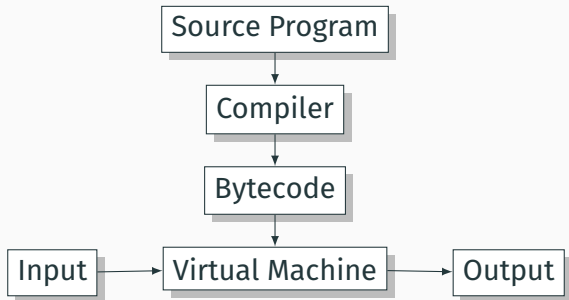
- **Pros:** source code distribution; short development cycle.

Interpreter

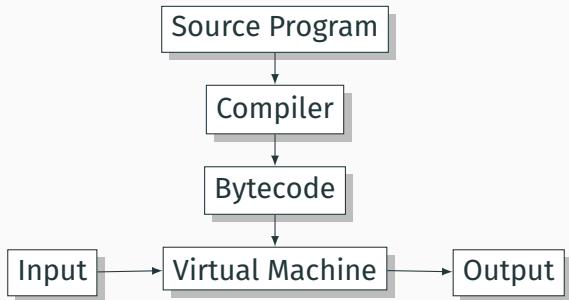


- **Pros:** source code distribution; short development cycle.
- **Cons:** translation is needed every time a statement is executed; lack optimization; map identifiers repeatedly.

Bytecode Interpreter

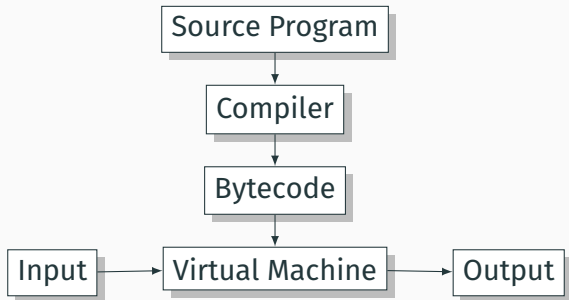


Bytecode Interpreter



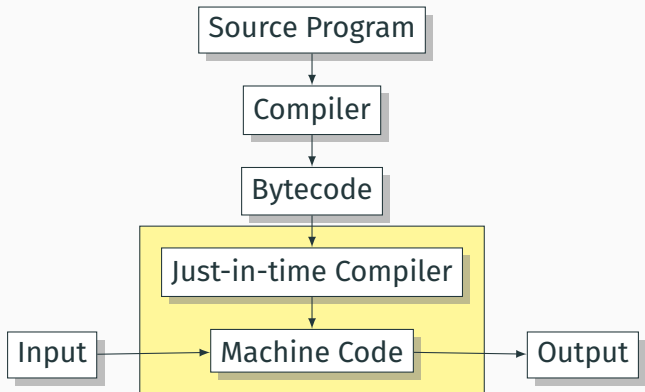
- **Pros:** bytecode is highly compressed and optimized; bytecode distribution.

Bytecode Interpreter



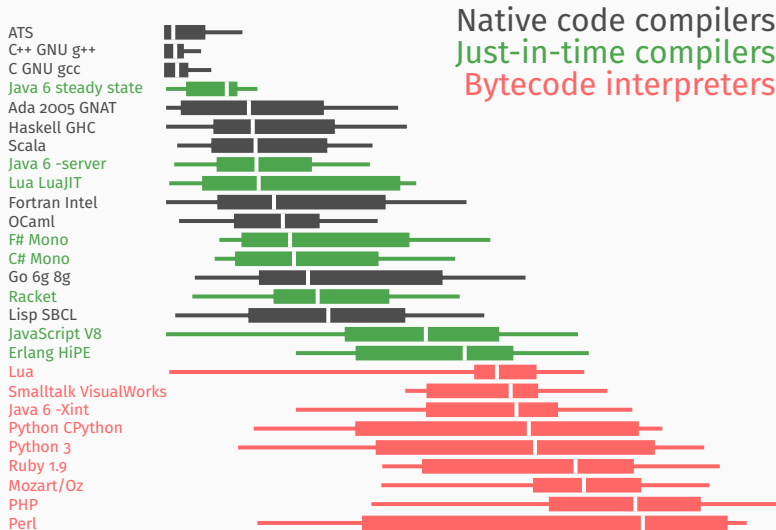
- **Pros:** bytecode is highly compressed and optimized; bytecode distribution.
- **Cons:** compilation overhead + interpreter overhead.

Just-In-Time Compiler



- **Pros:** compile and optimize many sections just before the execution; bytecode distribution.
- **Cons:** compilation overhead + warm-up overhead.

Language Speeds Compared



Source: <http://shootout.alioth.debian.org/>

Compilation Phases

Compiling a Simple Program

```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

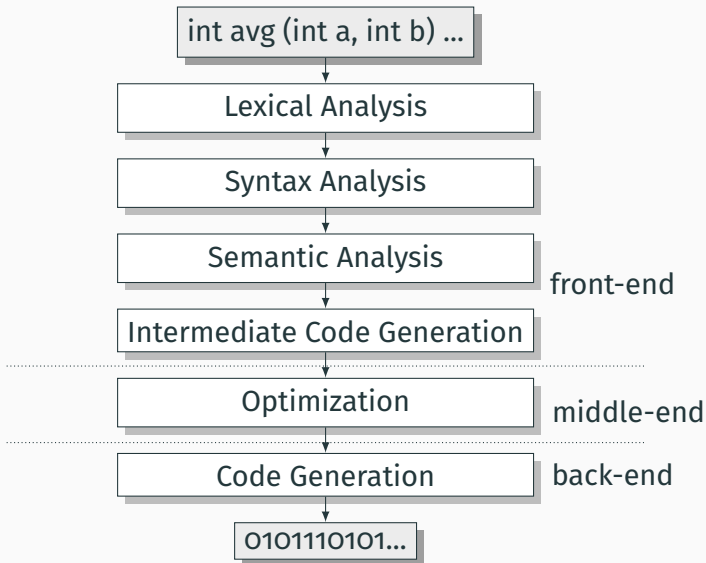
↓

Compiler

↓

0101110101...

Compilation Phases



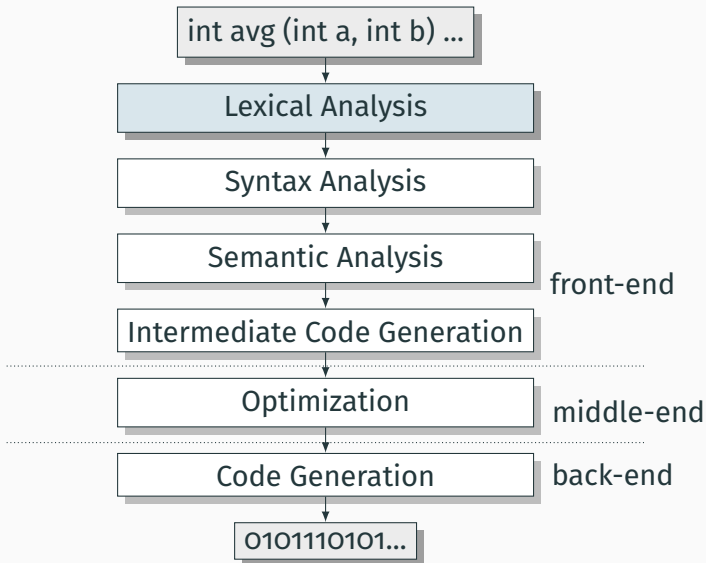
What the Compiler Sees

```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

i n t SP a v g (i n t SP a , SP i n t SP b) NL
{ NL
SP SP r e t u r n SP (a SP + SP b) SP / SP 2 ; NL
} NL

Just a sequence of characters

Lexical Analysis



Lexical Analysis Gives Tokens

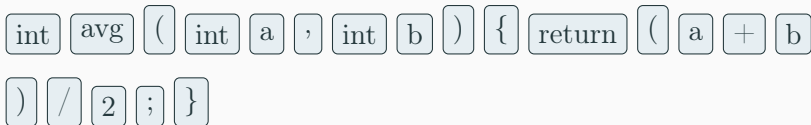
```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

int avg (int a , int b) { return (a + b) / 2 ; }

- A stream of **tokens**; whitespace, comments removed.

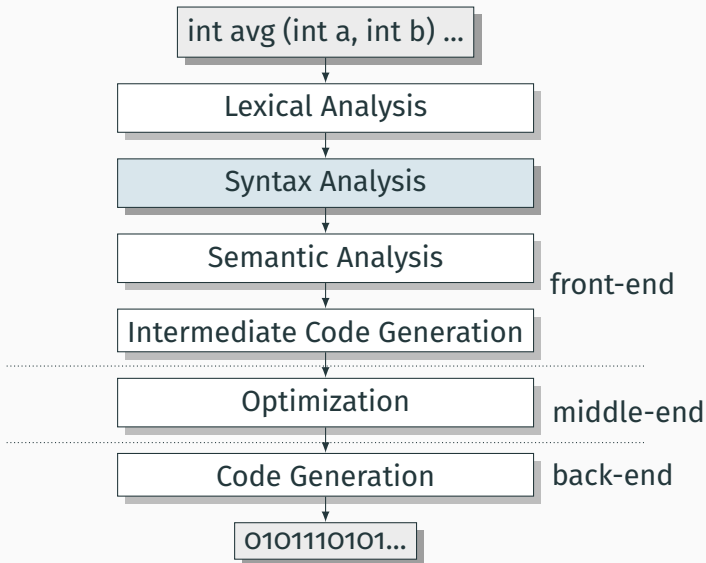
Lexical Analysis Gives Tokens

```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

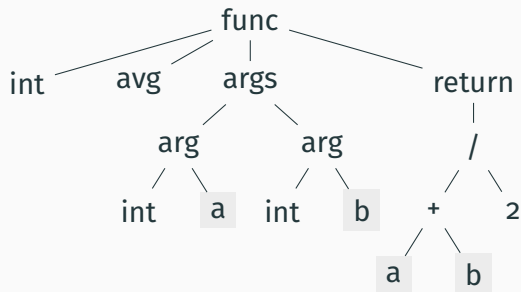


- A stream of **tokens**; whitespace, comments removed.
- Throw errors when failing to create tokens: malformed strings or numbers or invalid characters (such as non-ASCII characters in C).

Syntax Analysis



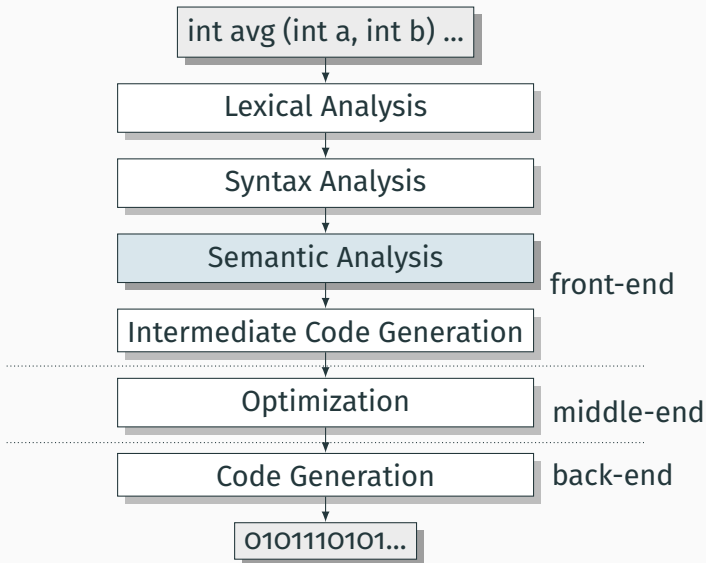
Syntax Analysis Gives an Abstract Syntax Tree



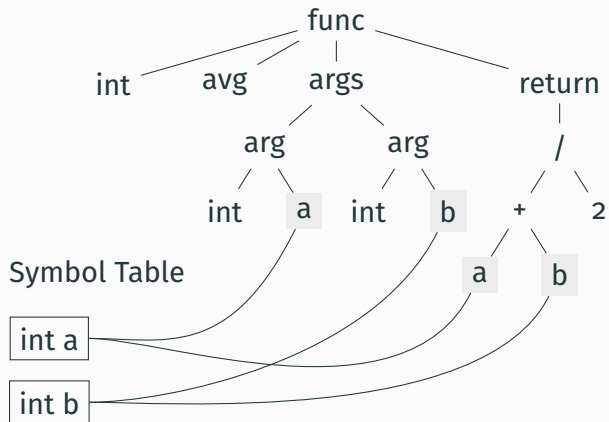
```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

- Syntax analysis will throw errors if “}” is missing. Lexical analysis will not.

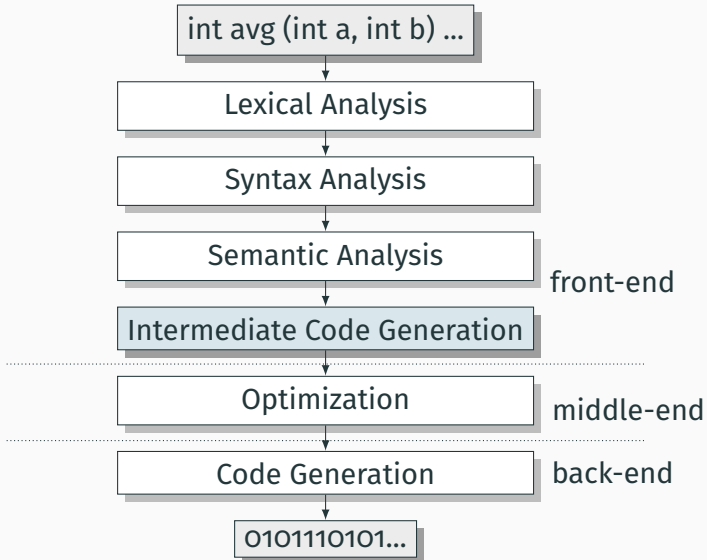
Semantic Analysis



Semantic Analysis: Resolve Symbols; Verify Types



Intermediate Code Generation



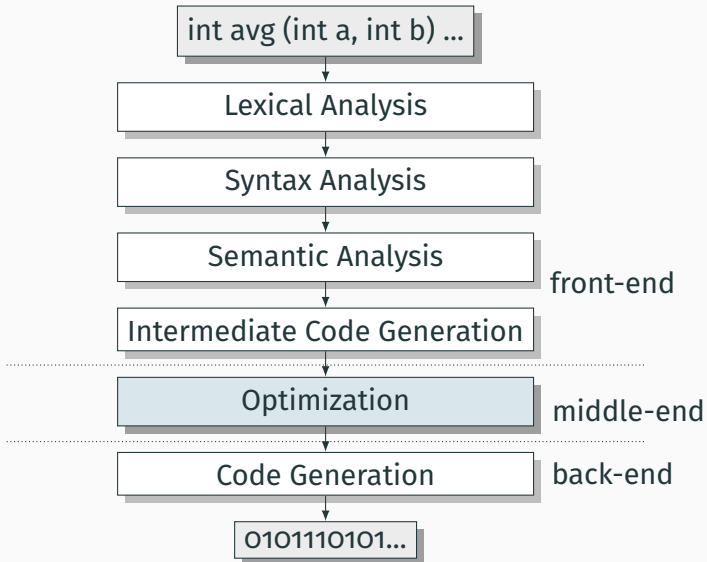
Translation into 3-Address Code

```
int avg(int a, int b)
{
    return (a + b) / 2;
}
```

Idealized assembly language w/ infinite registers

```
avg:
    t0 := a + b
    t1 := 2
    t2 := t0 / t1
    ret t2
```

Optimization



Optimization

```
avg:  
  t0 := a + b  
  t1 := 2  
  t2 := t0 / t1  
  ret t2
```

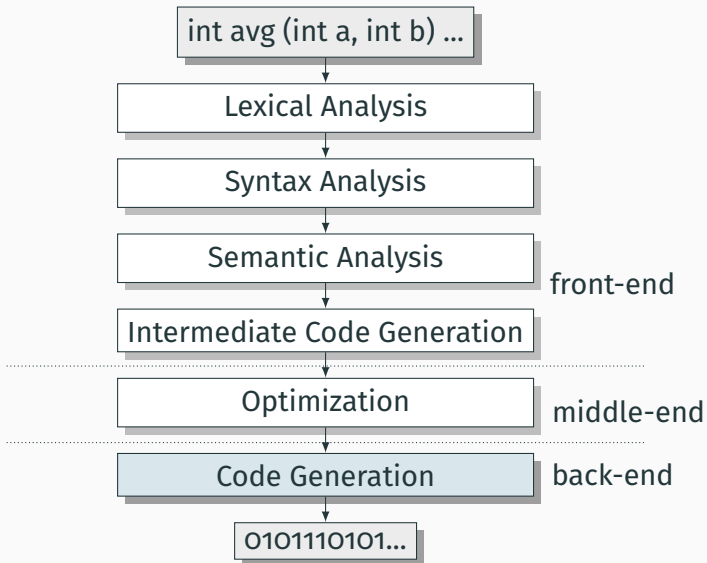
↓

Optimization

↓

```
avg:  
  t0 := a + b  
  t2 := t0 / 2  
  ret t2
```

Code Generation



Generation of x86 Assembly

```
avg:  
  t0 := a + b  
  t2 := t0 / 2  
  ret t2
```

Code Generation

```
avg:  pushl %ebp          # save BP  
      movl %esp,%ebp  
      movl 8(%ebp),%eax # load a from stack  
      movl 12(%ebp),%edx # load b from stack  
  
      addl %edx,%eax    # a += b  
      shr $1,%eax      # a /= 2  
      ret
```