

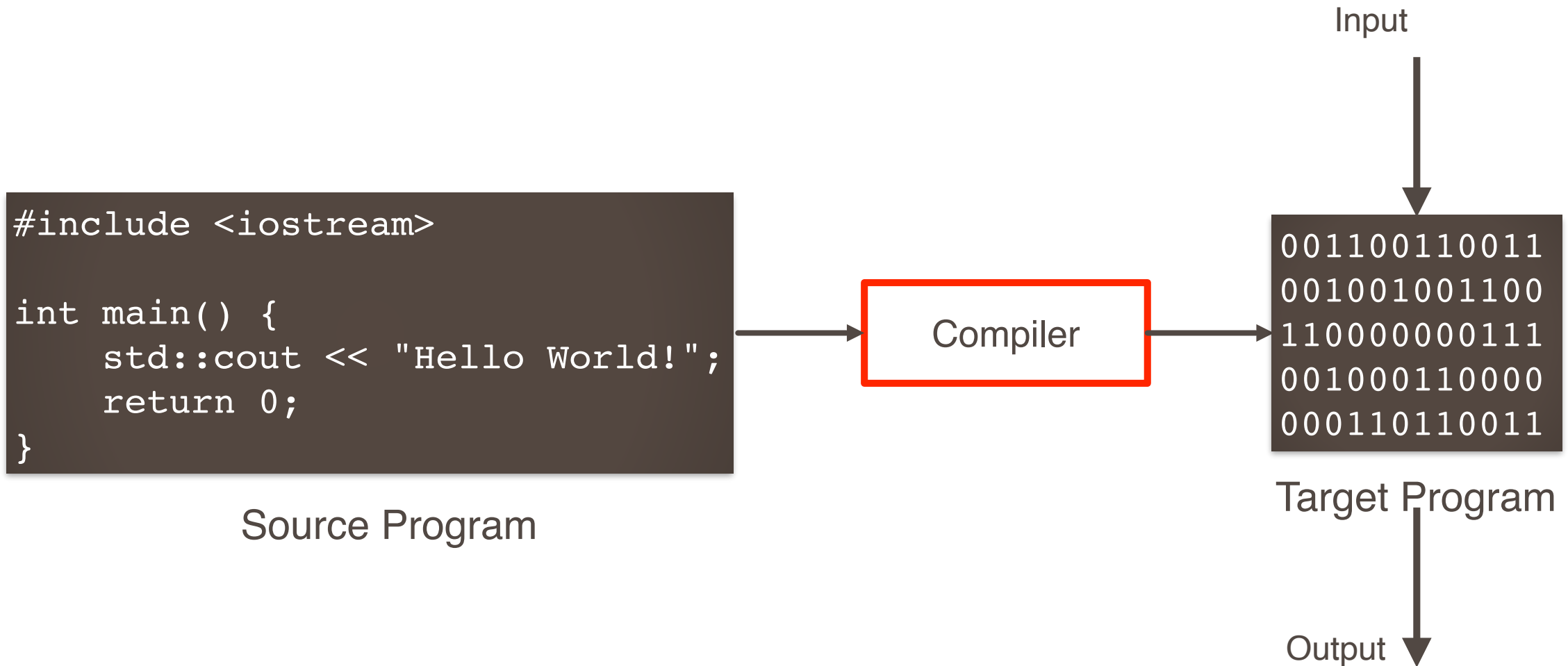
INTRODUCTION TO COMPILER

Baishakhi Ray

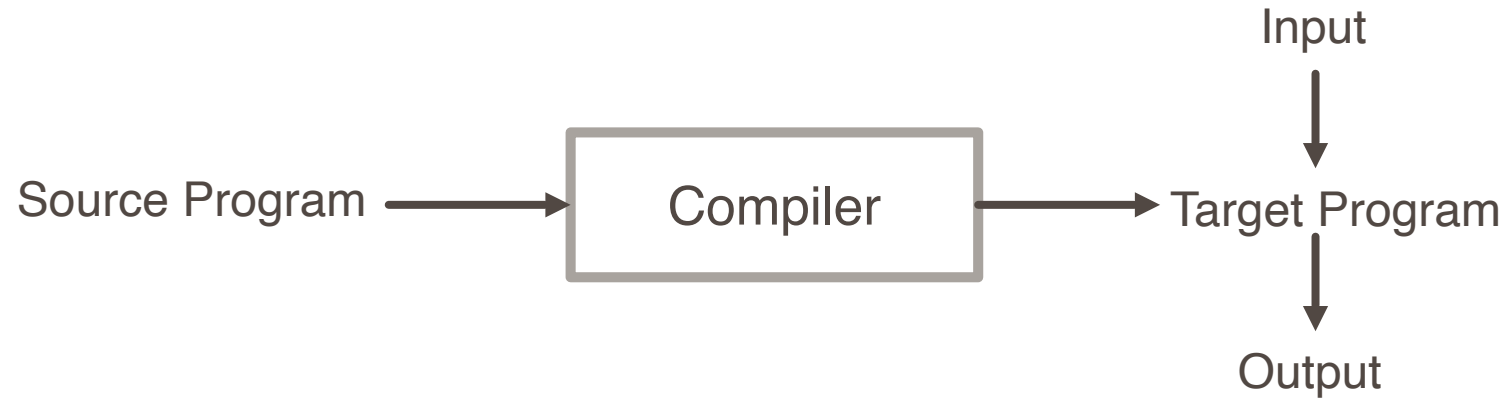
These slides are motivated from Prof. Calvin Lin, UT Austin



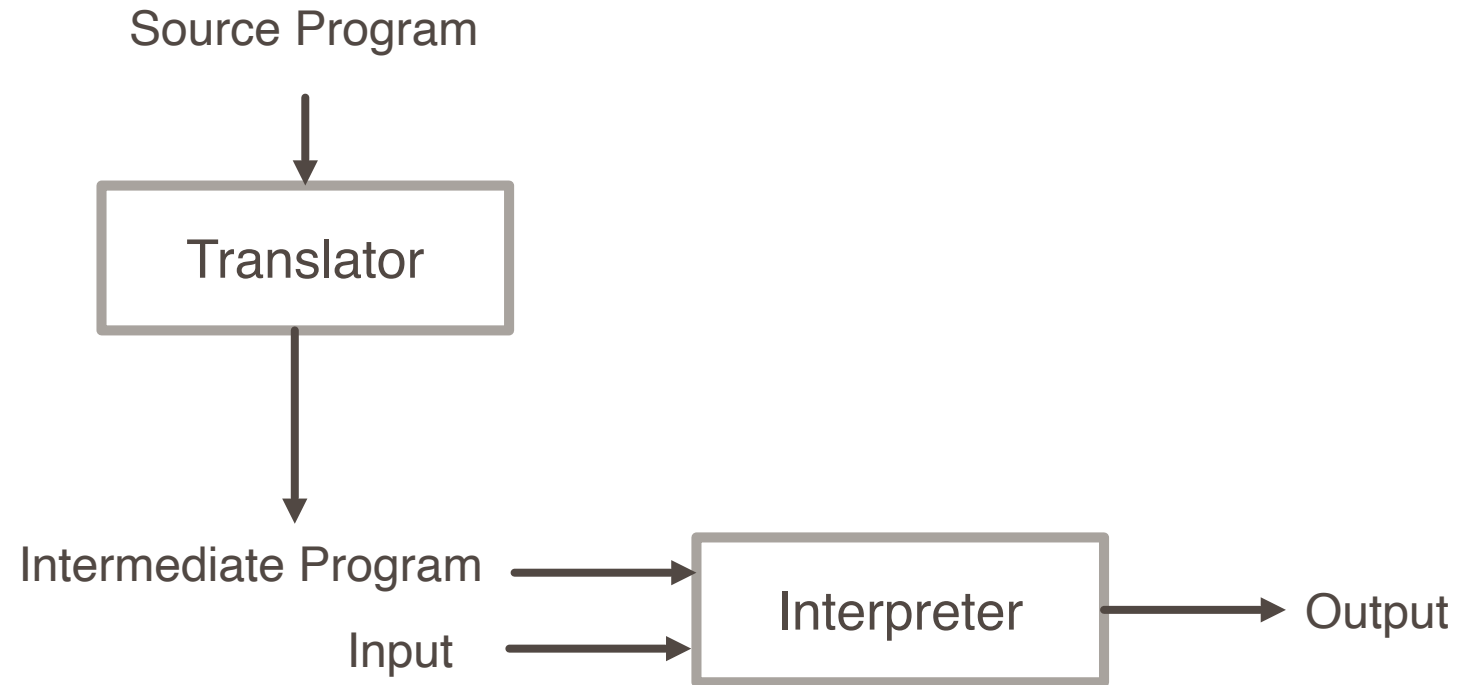
What is a Compiler?



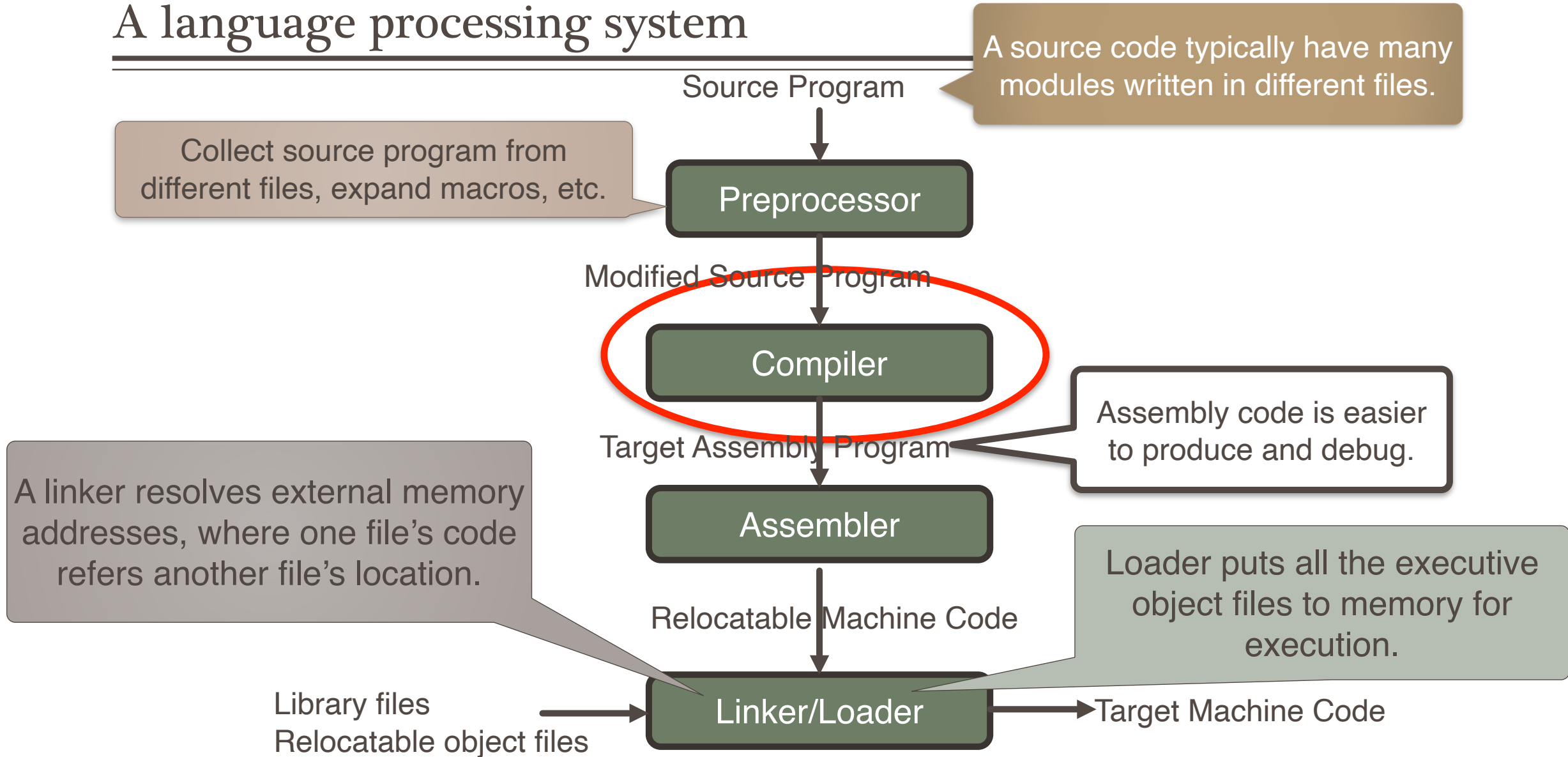
What is a Compiler?



A Hybrid Compiler



A language processing system



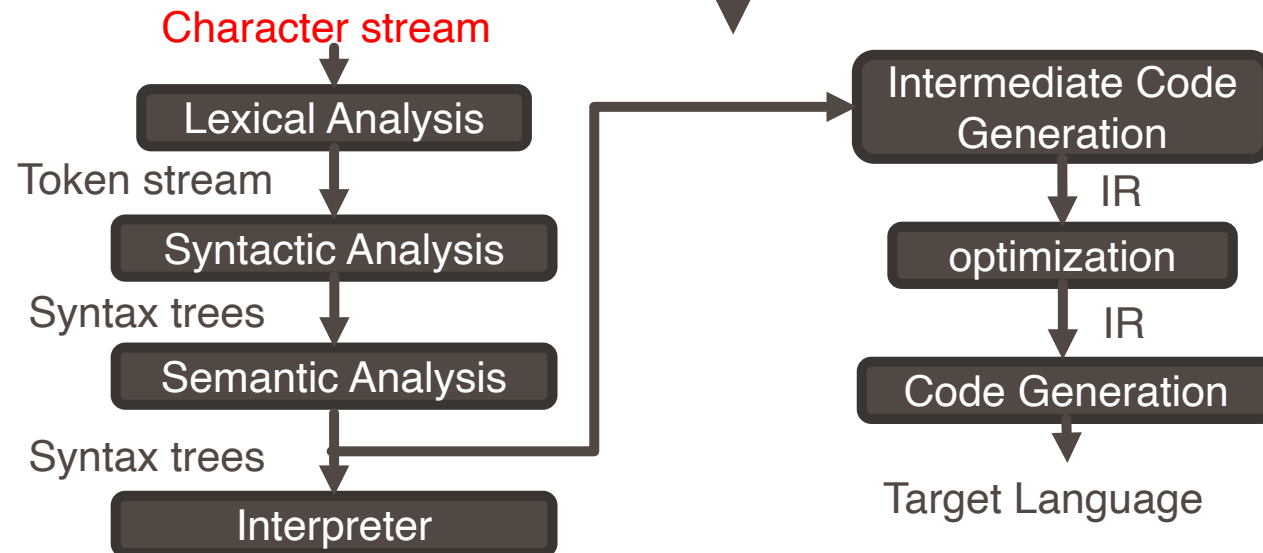
What is a Compiler?

```
#include <iostream>

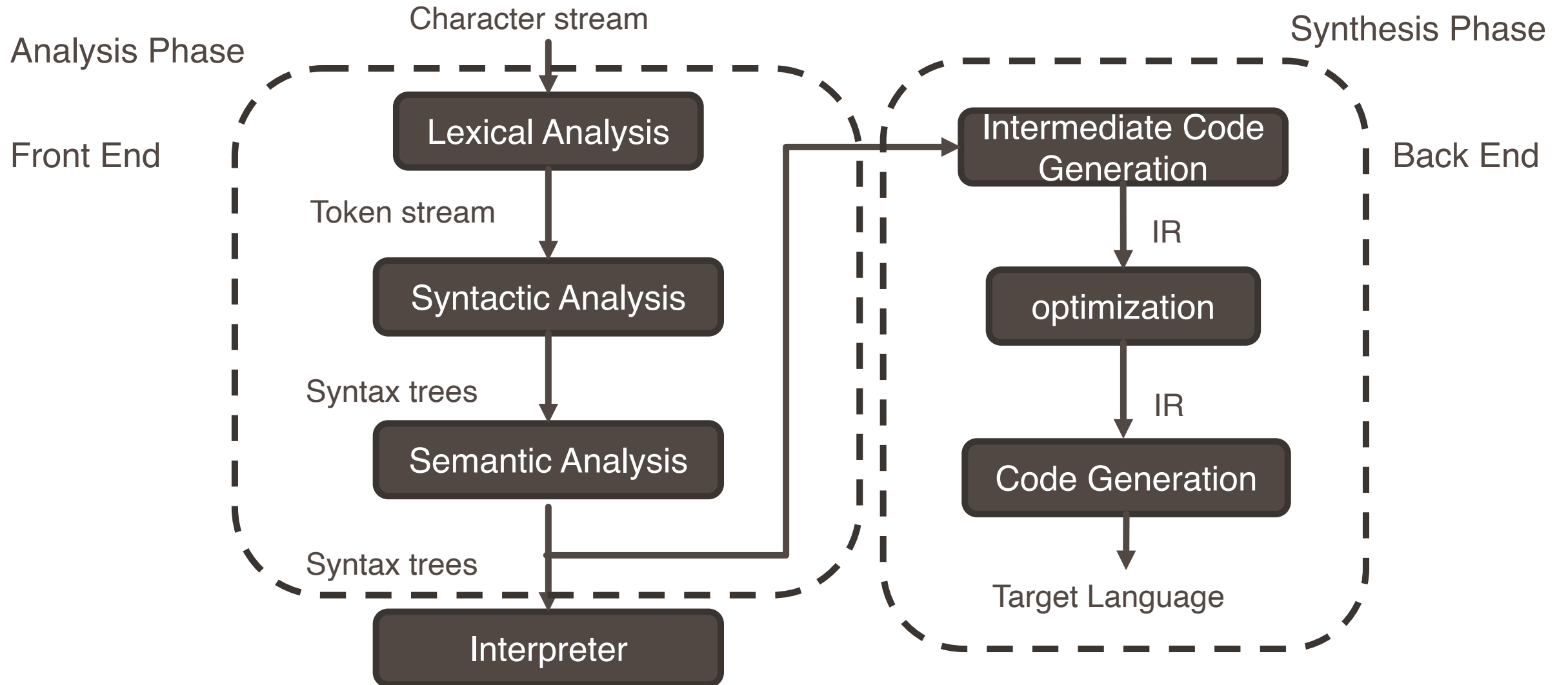
int main() {
    std::cout << "Hello World!";
    return 0;
}
```

Compiler

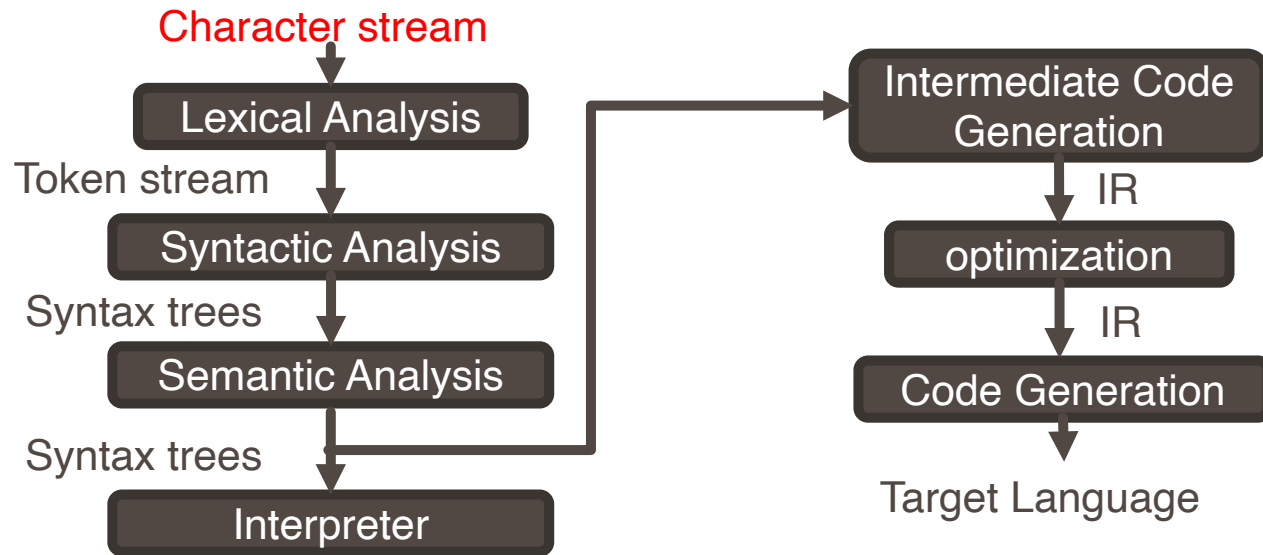
```
001100110011
001001001100
110000000111
001000110000
000110110011
```



Structure of a Typical Compiler



Input to Compiler

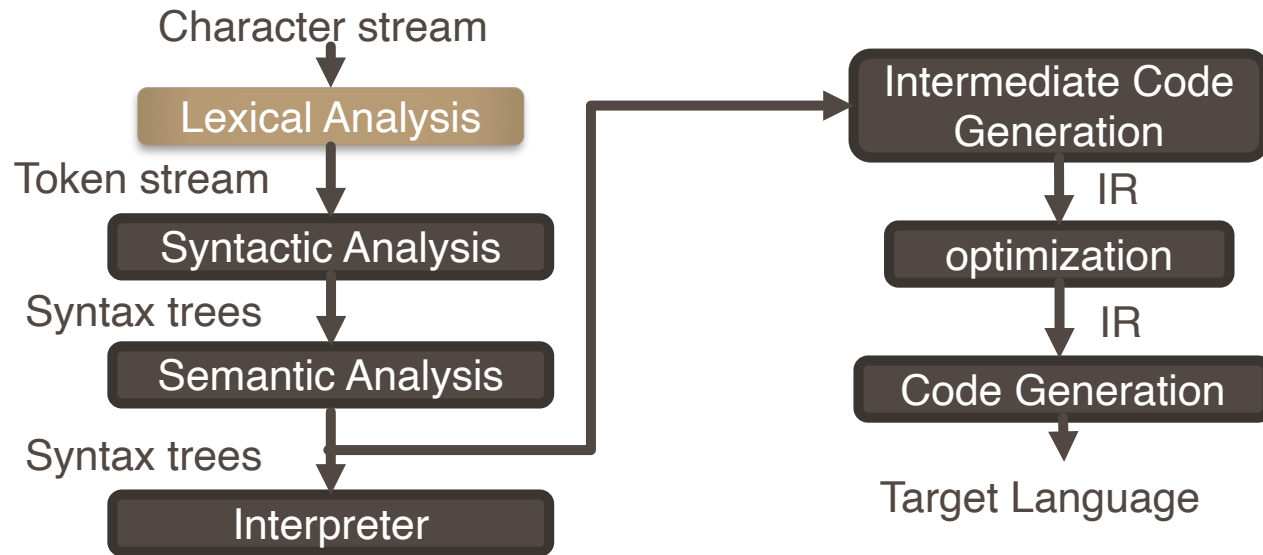


```
for i = 1 to 10 do  
    a[i] = x * 5;
```

```
for i = 1 to 10 do a[i] = x * 5;
```


Lexical Analysis

Break character stream into tokens (“words”)



```
for i = 1 to 10 do  
  a[i] = x * 5;
```

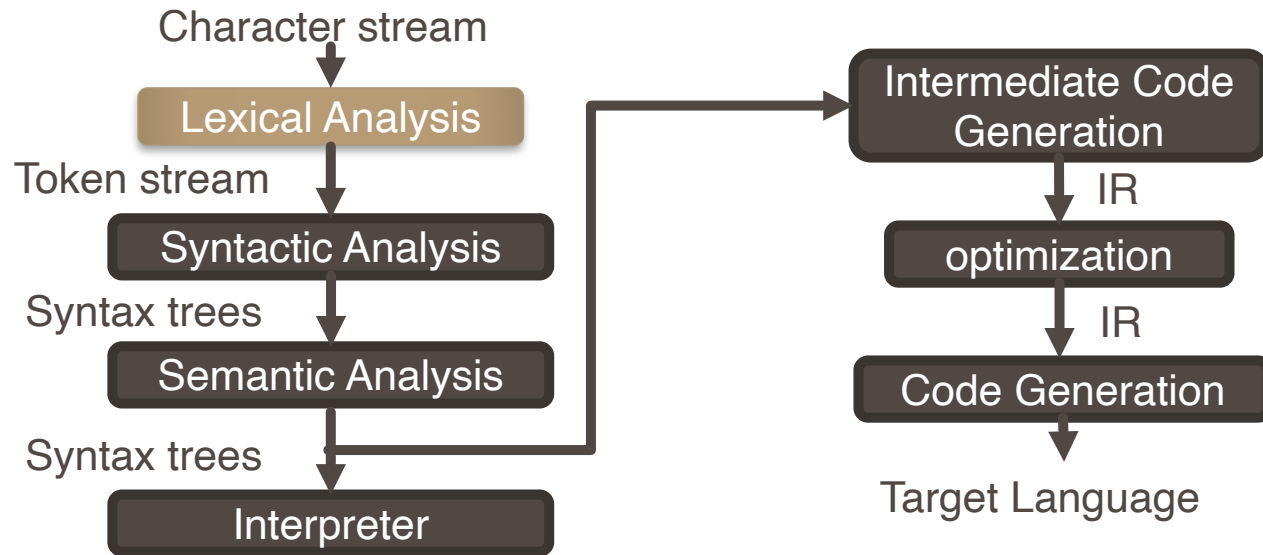
```
for id(i) <=> number(1) to number(10) do  
  id(a) <[> id(i) <]> <=> id(x) <*> number(5) <;>
```

Compiler Data Structure

- Symbol Tables
 - Compile-time data structures
 - Hold names, type information, and scope information for variables
- Scopes
 - A name space
 - e.g., In C, each set of curly braces defines a new scope
 - Can create a separate symbol table for each scope

Lexical Analysis

Break character stream into tokens (“words”)



```
for id(i) <=> number(1) to number(10) do
id(a) <[> id(i) <]> <=> id(x) <*> number(5) <;>
```

```
for <id,1> <=> number(1) to number(10) do
<id,2> <[> <id,1> <]> <=> <id,3> <*> number(5) <;>
```

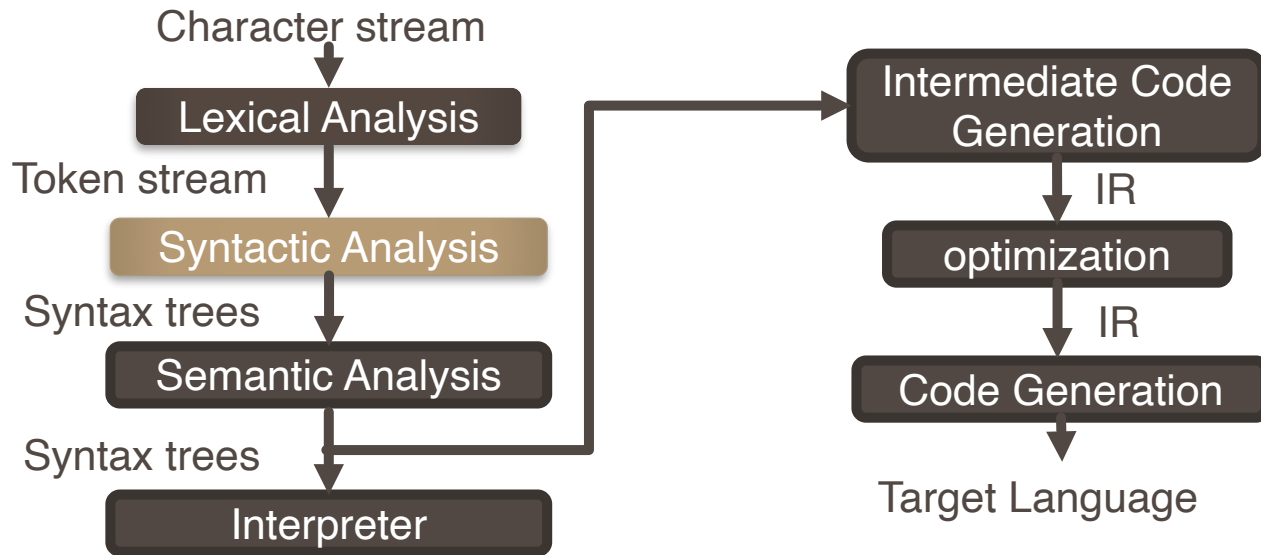
```
for i = 1 to 10 do
  a[i] = x * 5;
```

1	i	...
2	a	...
3	x	...

Symbol Table

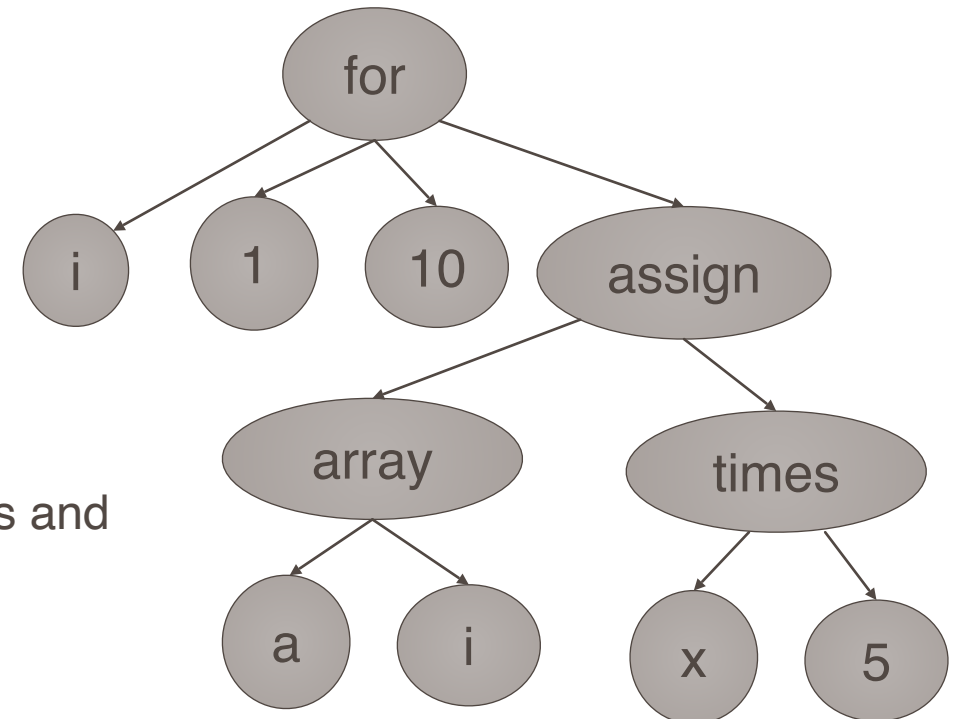
Syntactic Analysis (Parsing)

Impose Structure to Token Stream



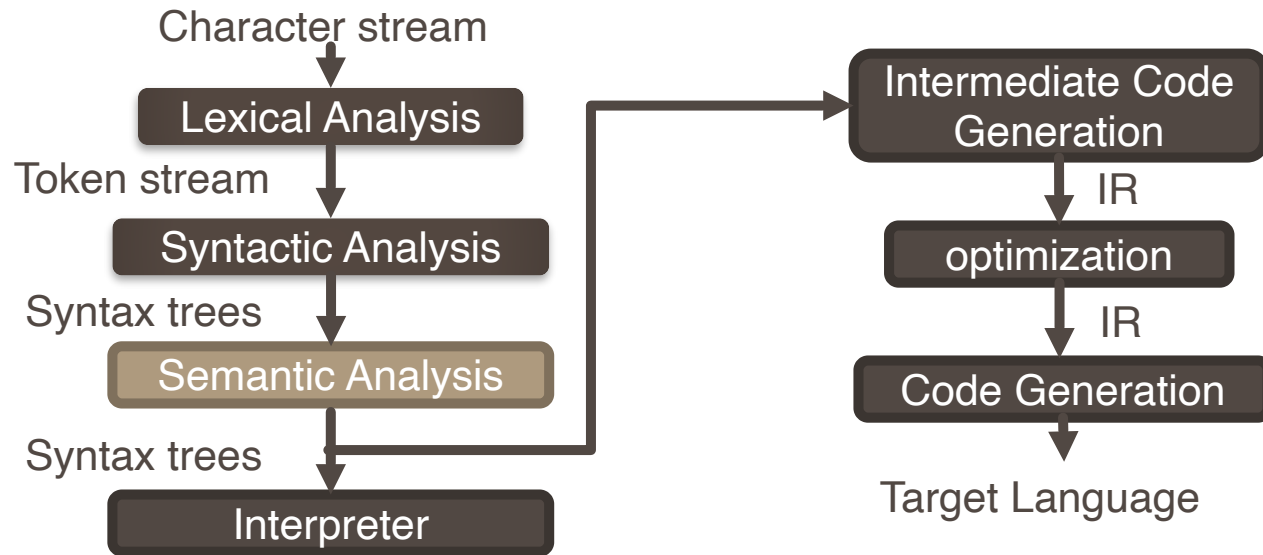
In a typical syntax tree, intermediate nodes represent operations and Leaf node represent the arguments of the operations.

```
for i = 1 to 10 do  
  a[i] = x * 5;
```



Semantic Analysis

Determine whether source is meaningful



```
for i = 1 to 10 do
  a[i] = x * 5;
```

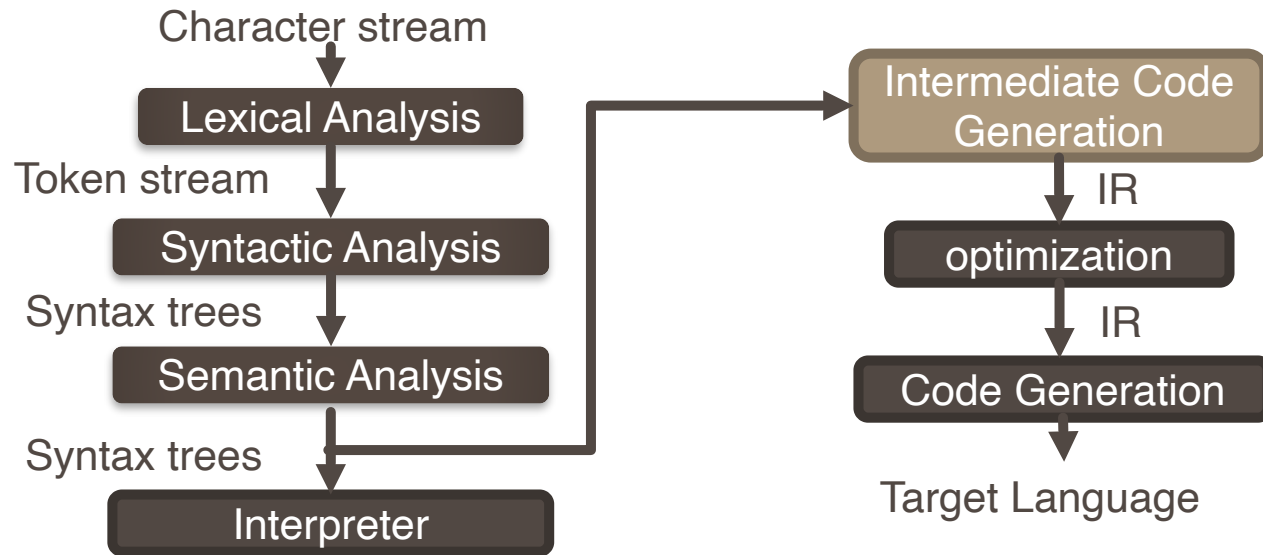
- Check for semantic errors
- Check for type errors
- Gather type information for subsequent stages
 - Relate variable uses to their declarations

Usage of Symbol Tables

- For each variable declaration:
 - Check for symbol table entry
 - Add new entry (parsing)
 - add type info (semantic analysis)
- For each variable use:
 - Check symbol table entry (semantic analysis)

Intermediate Code Generation

Transform AST into low-level intermediate representation (IR)

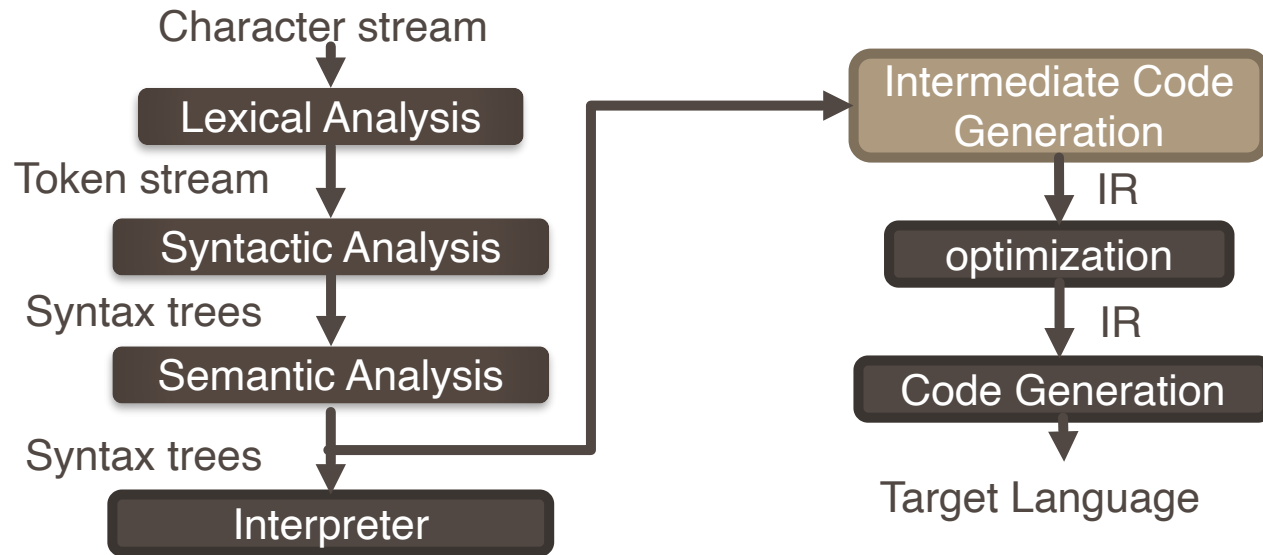


Simplifies the IR

- Removes high-level control structures:
 - for, while, do, switch
- Removes high-level data structures:
 - arrays, structs, unions, enums

Intermediate Code Generation

Transform AST into low-level intermediate representation (IR)



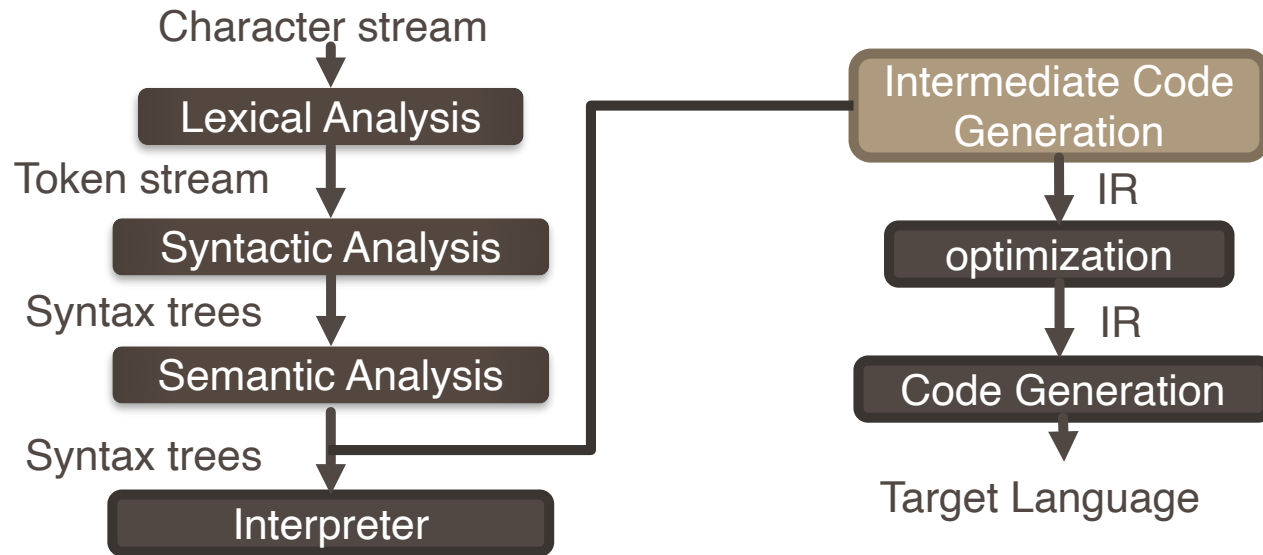
One possible result is assembly-like code

- Semantic lowering
- Control-flow expressed in terms of “gotos”
- Each expression is very simple (three-address code)

e.g., $x := a * b * c$

$t := a * b$
 $x := t * c$

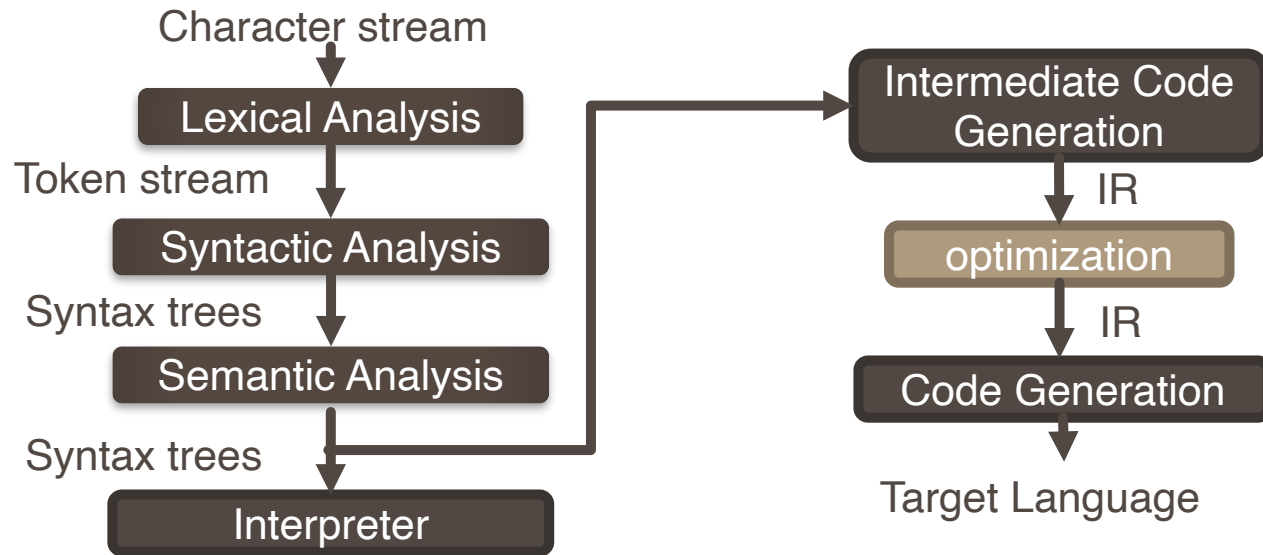
Intermediate Code Generation



```
for i = 1 to 10 do  
    a[i] = x * 5;
```

```
i := 1  
loop1:  
    t1 := x * 5  
    t2 := &a  
    t3 := sizeof(int)  
    t4 := t3 * i  
    t5 := t2 + t4  
    *t5 := t1  
    i := i + 1  
    if i <= 10 goto loop1
```

Optimization

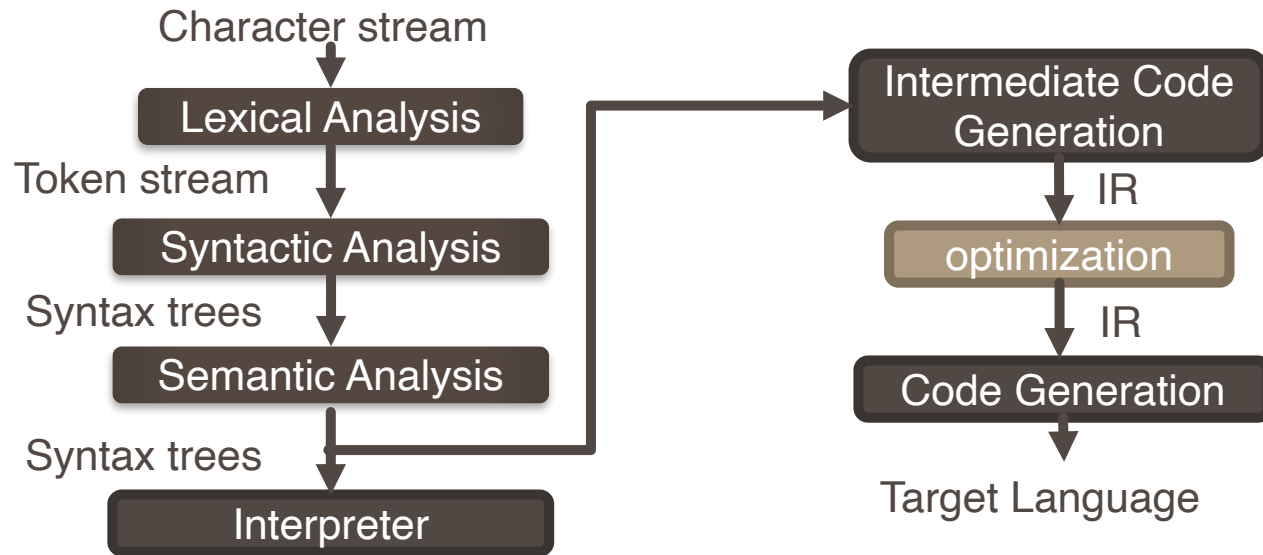


Mostly machine independent optimization
Phase aims to generate better code.

Better can be

- Faster
- Shorter
- Energy efficient
- ...

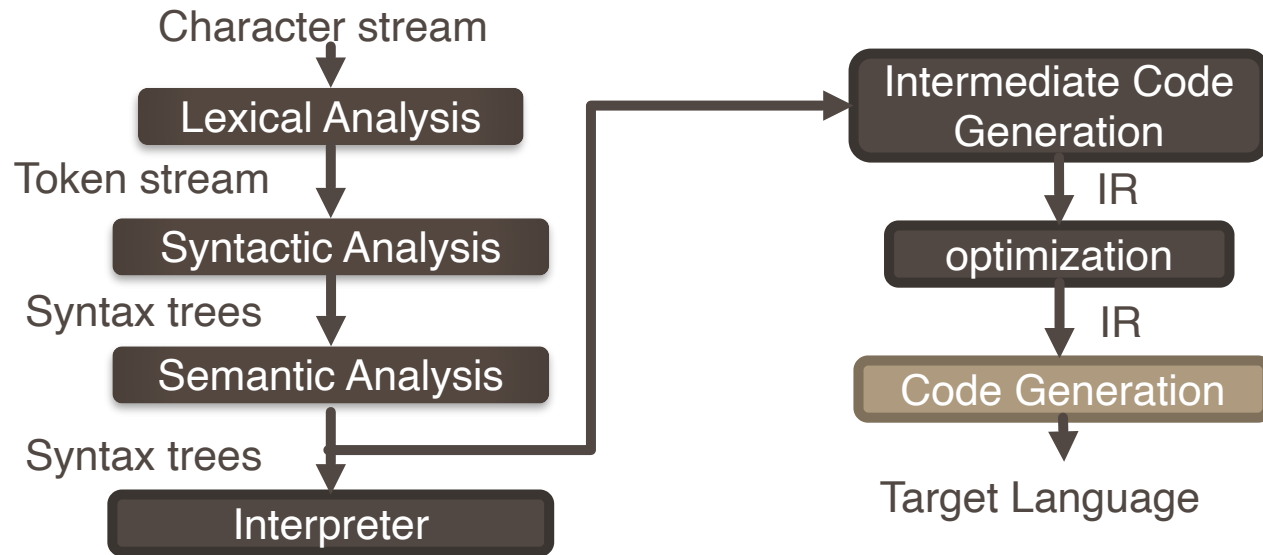
Optimization



```
for i = 1 to 10 do
  a[i] = x * 5;
```

```
i := 1
t3 := sizeof(int)
loop1:
  t1 := x * 5
  t2 := &a
  t3 := sizeof(int)
  t4 := t3 * i
  t5 := t2 + t4
  *t5 := t1
  i := i + 1
  if i <= 10 goto loop1
```

Low Level Code Generation



Register Transfer Language (RTL)

- Linear representation
- Typically language-independent
- Nearly corresponds to machine instructions

Example operations

- Assignment $x := y$
- Unary op $x := \text{op } y$
- Binary op $x := y \text{ op } z$
- Call $x := f()$
- Cbranch if $(x == 3) \text{ goto } L$
- Address of $p := \& y$
- Load $x := *(p+4)$
- Store $*(p+4) := y$

Exercise:

$$a = b + c * 5$$

Why studying compiler?

Isn't it a solved problem?

- Machines keep changing
 - New features present new problems (e.g., MMX, IA64, trace caches)
 - Changing costs lead to different concerns
- Languages keep changing
 - New ideas (e.g., OOP and GC) have gone mainstream
- Applications keep changing
 - Interactive, real-time, mobile, machine-learning based applications

Why studying compiler?

- Values keep changing
- We used to just care about run-time performance
- Now?
 - Compile-time performance
 - Code size
 - Correctness
 - Energy consumption
 - Security
 - Fault tolerance

Value added compilation

- The more we rely on software, the more we demand more of it
- Compilers can help— **treat code as data**
 - Analyze the code
- Correctness
- Security

Correctness and Security

- Can we check whether pointers and addresses are valid?
- Can we detect when untrusted code accesses a sensitive part of a system?
- Can we detect whether locks are used properly?
- Can we use compilers to certify that code is correct?
- Can we use compilers to verify that a given compiler transformation is correct?

Value-added Compilation

- The more we rely on software, the more we demand more of it
- Compilers can help— **treat code as data**
 - Analyze the code
 - Correctness
 - Security
 - Reliability
 - Program understanding
 - Program evolution
 - Software testing
 - Reverse engineering
 - Program obfuscation
 - Code compaction
 - Energy efficiency

Computation important → understanding computation important

Why studying compiler?

- Compilers are a fundamental building block of modern systems
- We need to understand their power and limitations
 - Computer architects
 - Language designers
 - Software engineers
 - OS/Runtime system researchers
 - Security researchers
 - Formal methods researchers (model checking, automated theorem proving)