Lecture Overview

1. Definition of Compiler
2. Typical Architecture of a Compiler
3. Example Analysis
4. Environment of a Compiler

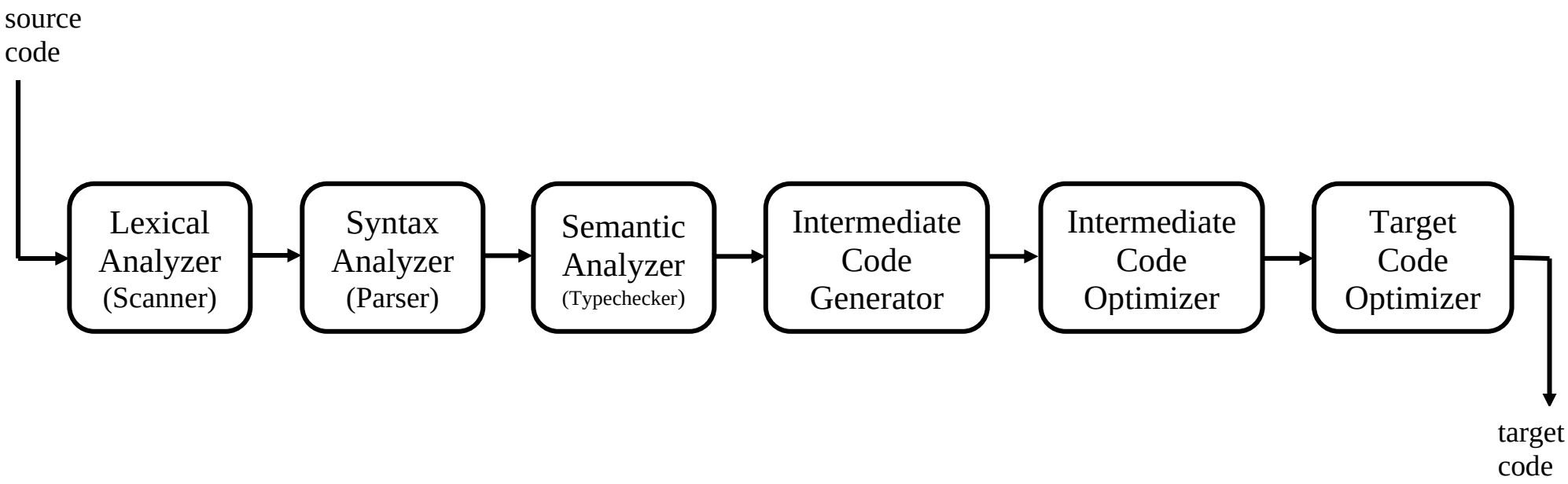
Compiler: a definition or two

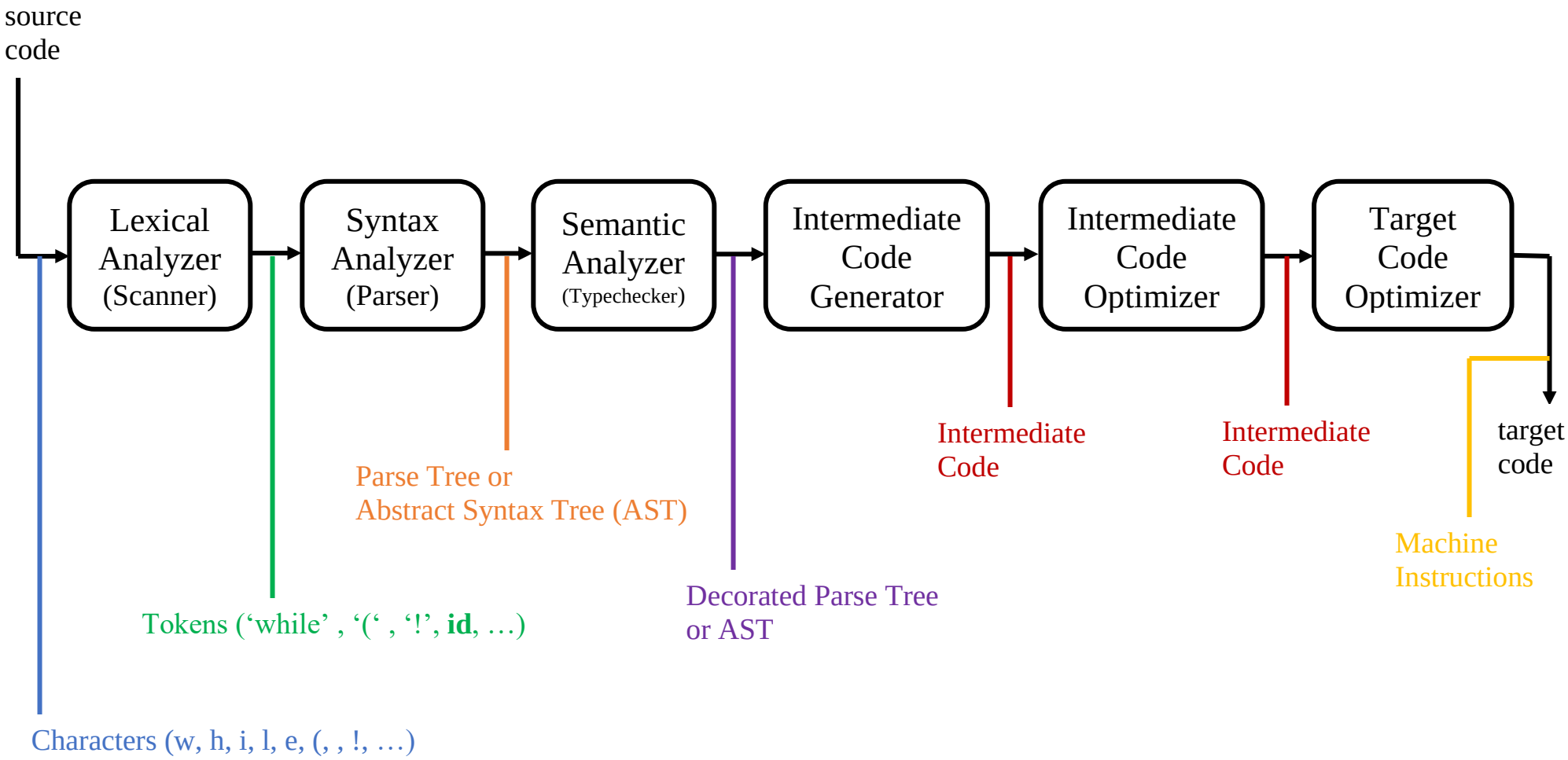
In general, a **compiler** is a program that converts one language to another. This is quite a broad definition. It includes natural-language translation, for instance, or a program to convert java to C++.

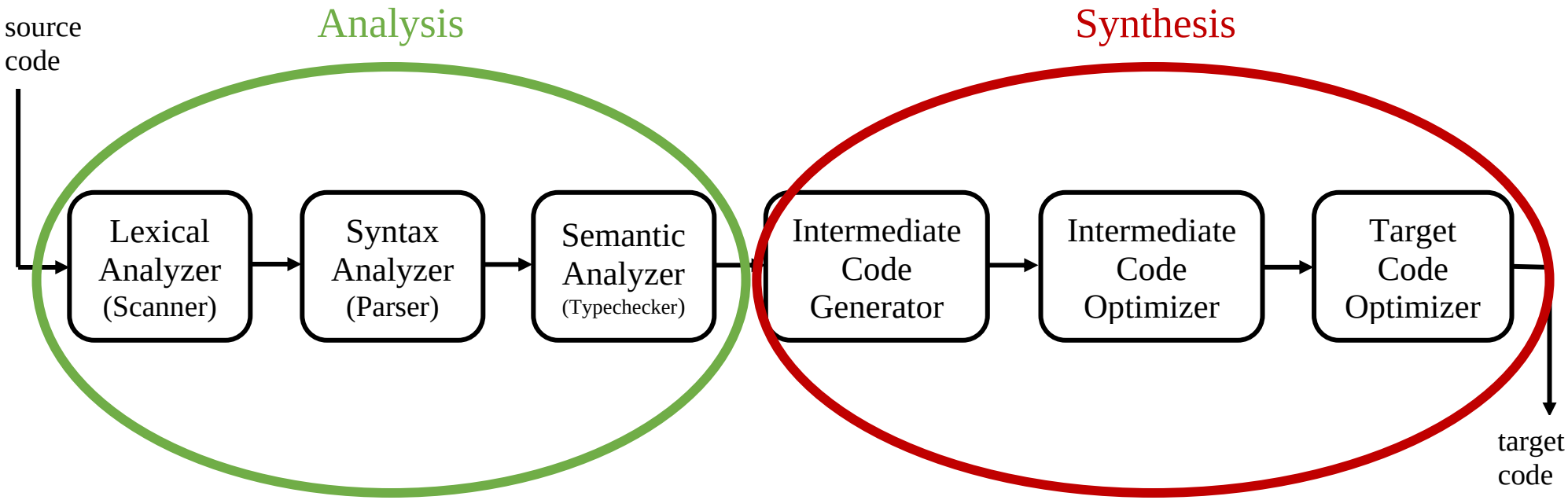
In formal language theory, a **language** is a set of strings. These strings are the valid sentences (say) in a natural language, or valid programs in a computer language.

A much more specific definition of compiler, and **the one we will use in this course**, is a program that converts from (a) a high-level general-purpose programming language to (b) a low-level machine or intermediate language.

A typical compiler architecture




source code
Lexical Analyzer (Scanner)
Syntax Analyzer (Parser)
Semantic Analyzer (Typechecker)
Intermediate Code Generator
Intermediate Code Optimizer
Target Code Optimizer
target code
Characters (w, h, i, l, e, (, , !, …)
Tokens (‘while’ , ‘(‘ , ‘!’, id, …)
Parse Tree or Abstract Syntax Tree (AST)
Decorated Parse Tree or AST
Intermediate Code
Intermediate Code
Machine Instructions


source
code
Analysis
Synthesis
Lexical
Analyzer
(Scanner)
Syntax
Analyzer
(Parser)
Semantic
Analyzer
(Typechecker)
Intermediate
Code
Generator
Intermediate
Code
Optimizer
Target
Code
Optimizer
target
code

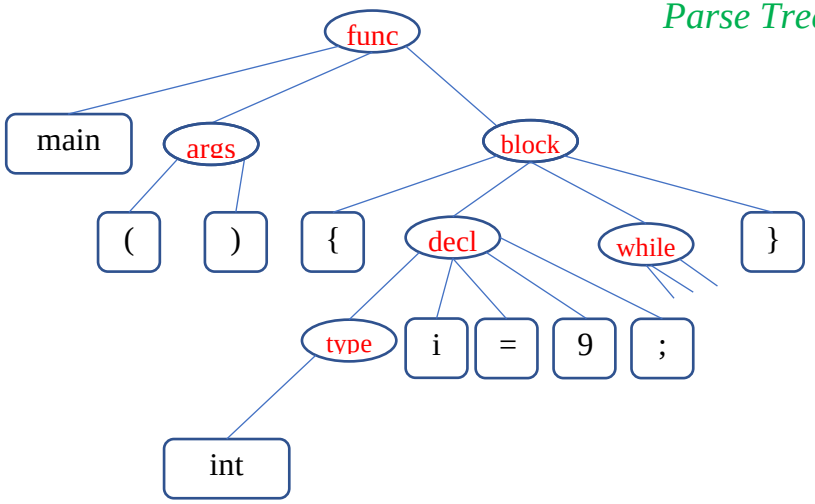
```
main() {
   int i = 9;
   while( i != 1 ) {
      printf(“%d\n”, i);
      i = syracuse(i);
    }
}

int syracuse(int n) {
   if(n % 2 == 0)
      return n / 2;
   else
      return 3 * n + 1;
}
```

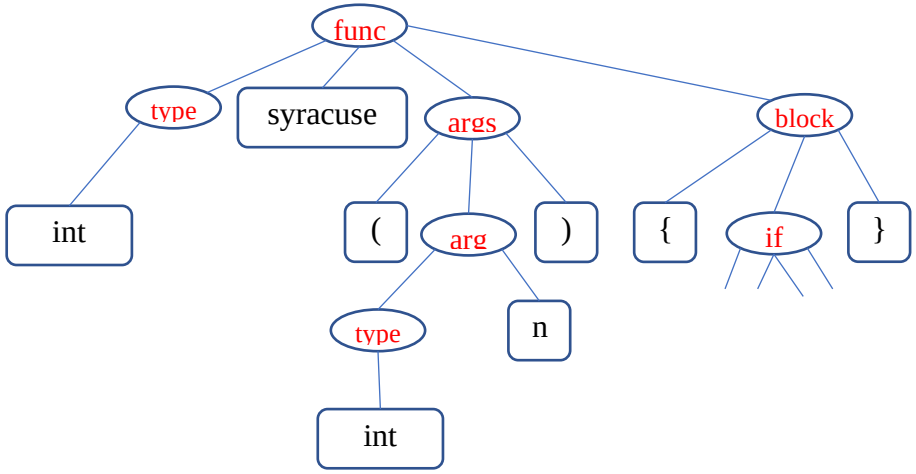
characters
m
a
i
n
(
)
space
{
newline
space
space
space
i
n
t
space
i
space
=
space
9
;
newline
space
space
space
w
h
i
l
e
(
space
i
space
!
=
…

tokens
main
(
)
{
int
i
=
9
;
while
(
i
!=
1
)
{
printf
(
“%d\n”
,
i
)
;
i
=
syracuse
(
i
)
;
}
}
int
syracuse
(
int
…

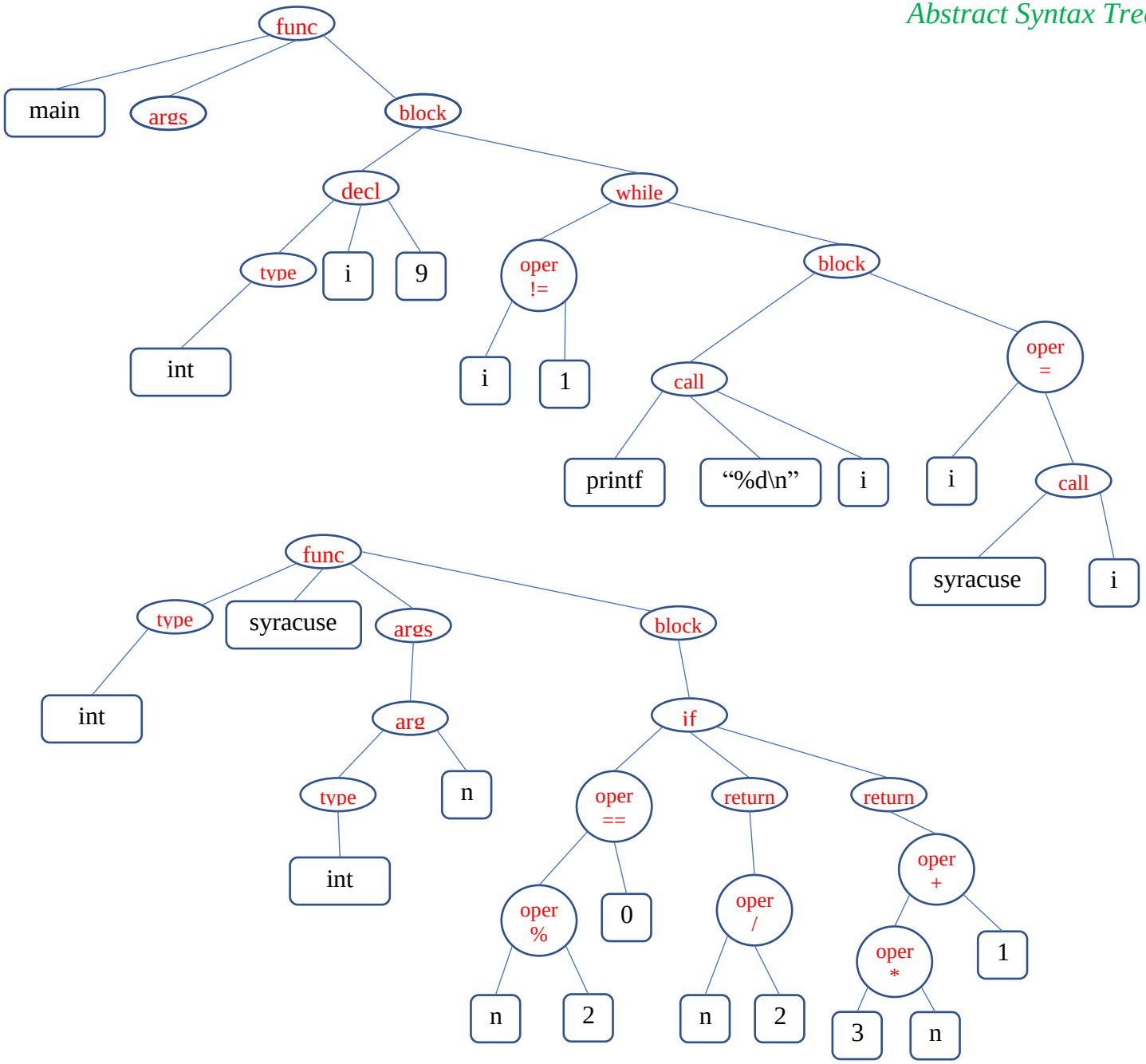
```
main() {
   int i = 9;
   while( i != 1 ) {
      printf("%d\n", i);
      i = syracuse(i);
    }
}

int syracuse(int n) {
   if(n % 2 == 0)
      return n / 2;
   else
      return 3 * n + 1;
}
```

Parse Tree



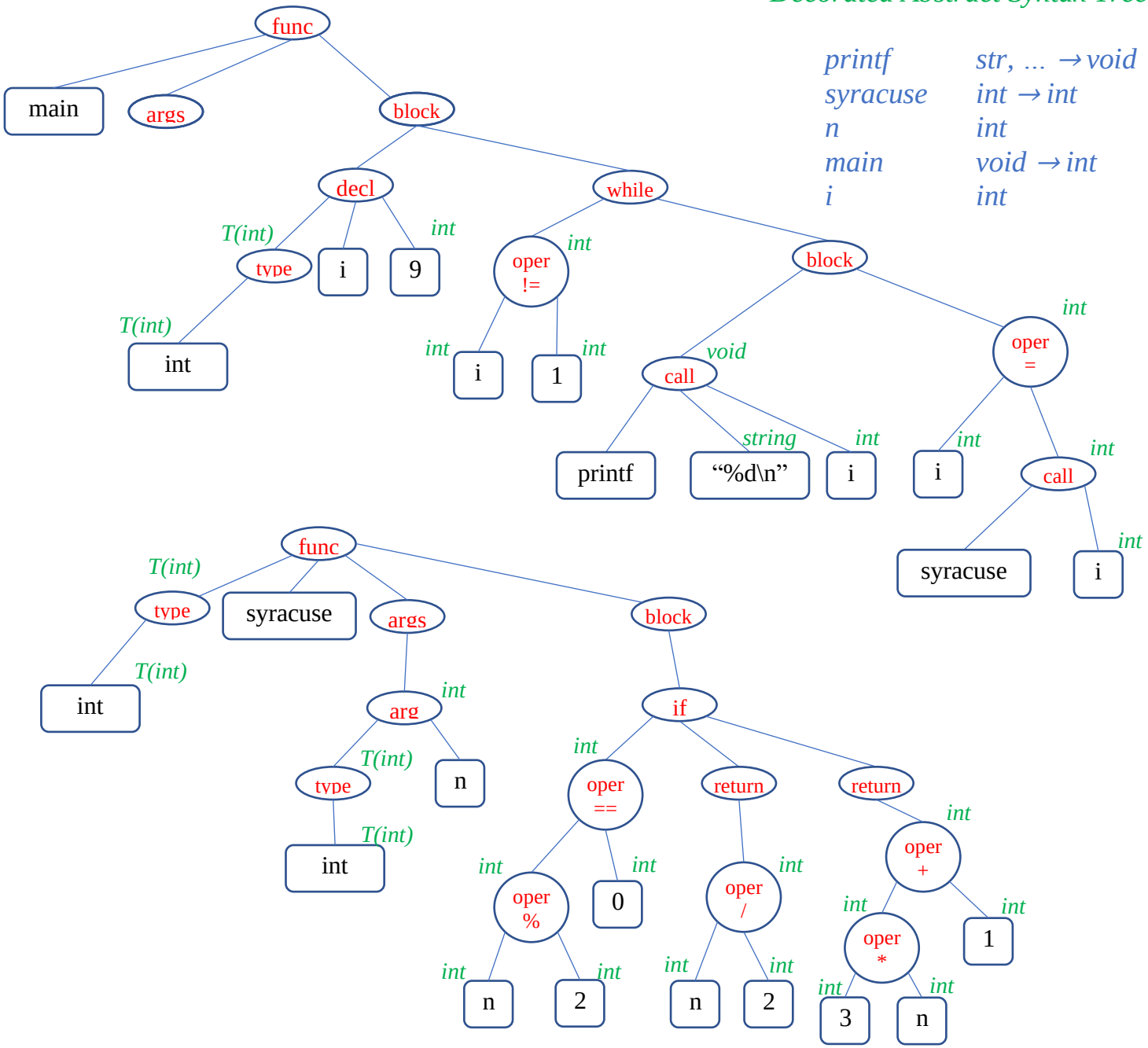


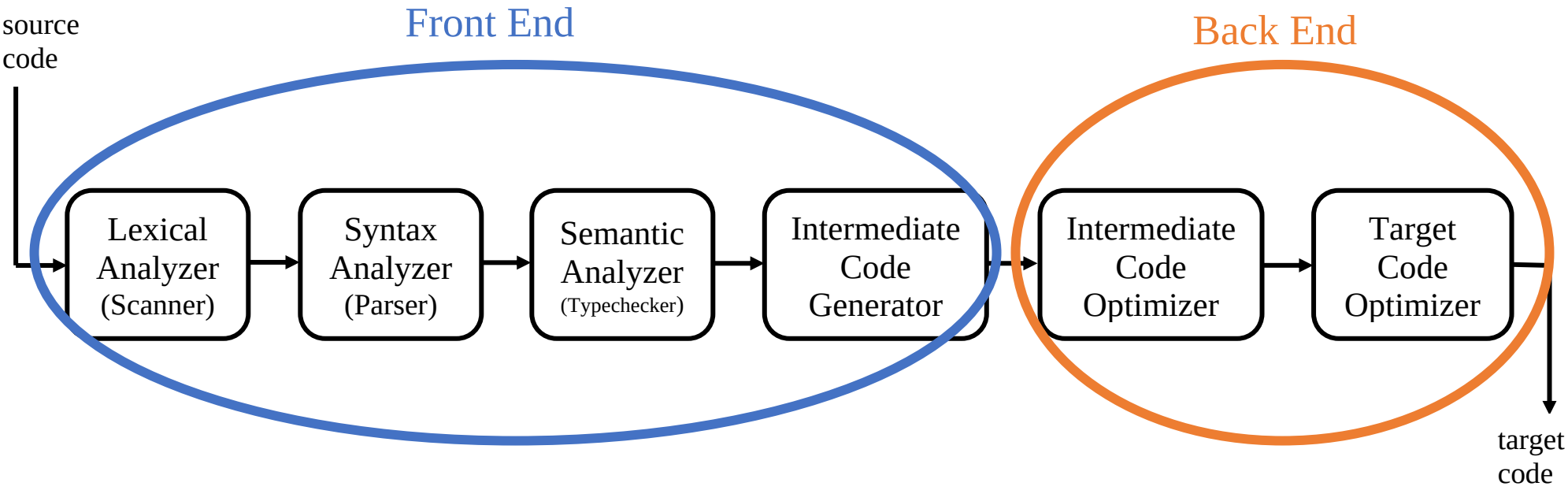
Abstract Syntax Tree

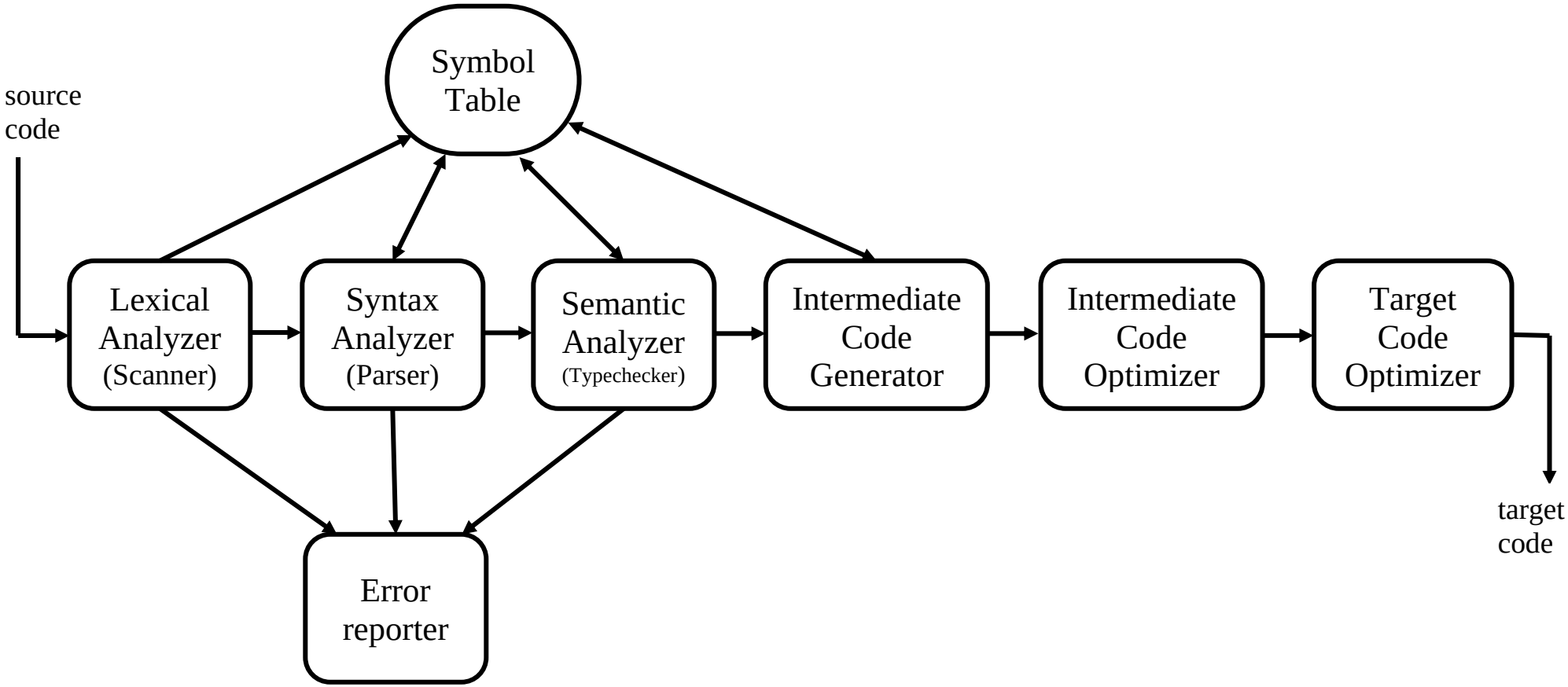


Decorated Abstract Syntax Tree

printf	*str, ... → void*
syracuse	*int → int*
n	*int*
main	*void → int*
i	*int*




source
code
Front End
Back End
Lexical
Analyzer
(Scanner)
Syntax
Analyzer
(Parser)
Semantic
Analyzer
(Typechecker)
Intermediate
Code
Generator
Intermediate
Code
Optimizer
Target
Code
Optimizer
target
code


Symbol Table
source code
Lexical Analyzer (Scanner)
Syntax Analyzer (Parser)
Semantic Analyzer (Typechecker)
Intermediate Code Generator
Intermediate Code Optimizer
Target Code Optimizer
target code
Error reporter

The environment of a compiler

An **interpreter** is like a compiler, except that it executes the actions specified by the source program rather than creating target code. Interpreted programs tend to be slower than compiled ones.

An **assembler** is a (broad-sense) compiler that translates from an assembly language to machine language.

A **linker** is a program that takes a target-language portion of a program and combines it with other portions of the program that were compiled separately. These separately-compiled portions include **libraries**.

A **loader** is a program that takes a target-language program out of a file and places it into the memory of a computer, then starts it running.

A **debugger** is a program that allows a programmer to step through their program and inspect what is happening along the way.

An **integrated development environment (IDE)** is typically a compiler, linker, loader, debugger, and an editor combined and working with each other.

When something happens while the compiler is running, we call it compile-time or **static**.

When something happens while the program the compiler produced is running, we call it run-time or **dynamic**.

There are -time terms for linking and loading, too: link-time and load-time.

For example, we may link a program together with all its parts and libraries right after the compiler finishes (often as part of the compiler) and this is called **static linking.**

If we link the program together when loading it, it's **load-time linking.**

If we link parts of the program together while the program is running, it's **dynamic linking.** This is accompanied by **dynamic loading**, which pulls in some parts of the program off of a long-term storage medium while the program is running.