UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

Compilers: Course Organisation & Introduction

Alessandra Di Pierro
alessandra.dipierro@univr.it

Dipartimento di Informatica
Università di Verona

Learning About Compilers

You will hardly ever be asked to write a compiler for a general-purpose language, so...

Learning About Compilers

You will hardly ever be asked to write a compiler for a general-purpose language, so...

... why should one attend this course?

Learning About Compilers

You will hardly ever be asked to write a compiler for a general-purpose language, so...
... why should one attend this course?

- Competence of a computer scientist (a good craftman...)
  - See theory come to life
  - Learn how to build programming languages
  - Learn how programming languages work
  - Learn tradeoffs in language design.

Learning About Compilers

You will hardly ever be asked to write a compiler for a general-purpose language, so...
... why should one attend this course?

- Competence of a computer scientist (a good craftman...)
  - See theory come to life
  - Learn how to build programming languages
  - Learn how programming languages work
  - Learn tradeoffs in language design.
- Techniques useful for other purposes as well

Learning About Compilers

You will hardly ever be asked to write a compiler for a general-purpose language, so...
... why should one attend this course?

- Competence of a computer scientist (a good craftman...)
  - See theory come to life
  - Learn how to build programming languages
  - Learn how programming languages work
  - Learn tradeoffs in language design.
- Techniques useful for other purposes as well
- Compilers and interpreter for DSL (Domain Specific Languages)

Learning About Compilers

You will hardly ever be asked to write a compiler for a general-purpose language, so...
... why should one attend this course?

- Competence of a computer scientist (a good craftman...)
  - See theory come to life
  - Learn how to build programming languages
  - Learn how programming languages work
  - Learn tradeoffs in language design.
- Techniques useful for other purposes as well
- Compilers and interpreter for DSL (Domain Specific Languages)

... and because it's a beautiful subject!

1. Course Organisation
   - Reading Material

2. Compilation: From Source to Machine Code
   - Another Simple Introductory Example
   - Compilation Phases

Course Format: Lectures and Timetable

- Two weekly lectures:
  - Mondays: 14:30–17:30
  - Thursdays: 11:30–13:30
- Exercise Session:
  - Weekly on Thursdays (11:30–13:30)
  - Complementing exercises for the lectures
- Lab Session:
  - Thursdays : 16:30–19:30
- Exam:
  - written exam

Reading Material

- Mogensen T. A. (2011),
  Introduction to Compiler Design, Springer, London.
- Mogensen T. A. (2000–2010), *Basics of Compiler Design*,
  DIKU, self published:
  `http://www.diku.dk/~torbenm/Basics/`
- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman (2007), *Compilers: Principles, Techniques, and Tools* (aka Dragon Book: `http://dragonbook.stanford.edu`)
- Patterson, D. A. and Hennessy, J. L. (1998).
  Computer Organization & Design, the Hardware/Software Interface, Morgan Kaufmann. Appendix A freely available at
  `http://www.cs.wisc.edu/~larus/HP_AppA.pdf`

1 Course Organisation
- Reading Material

2 Compilation: From Source to Machine Code
- Another Simple Introductory Example
- Compilation Phases

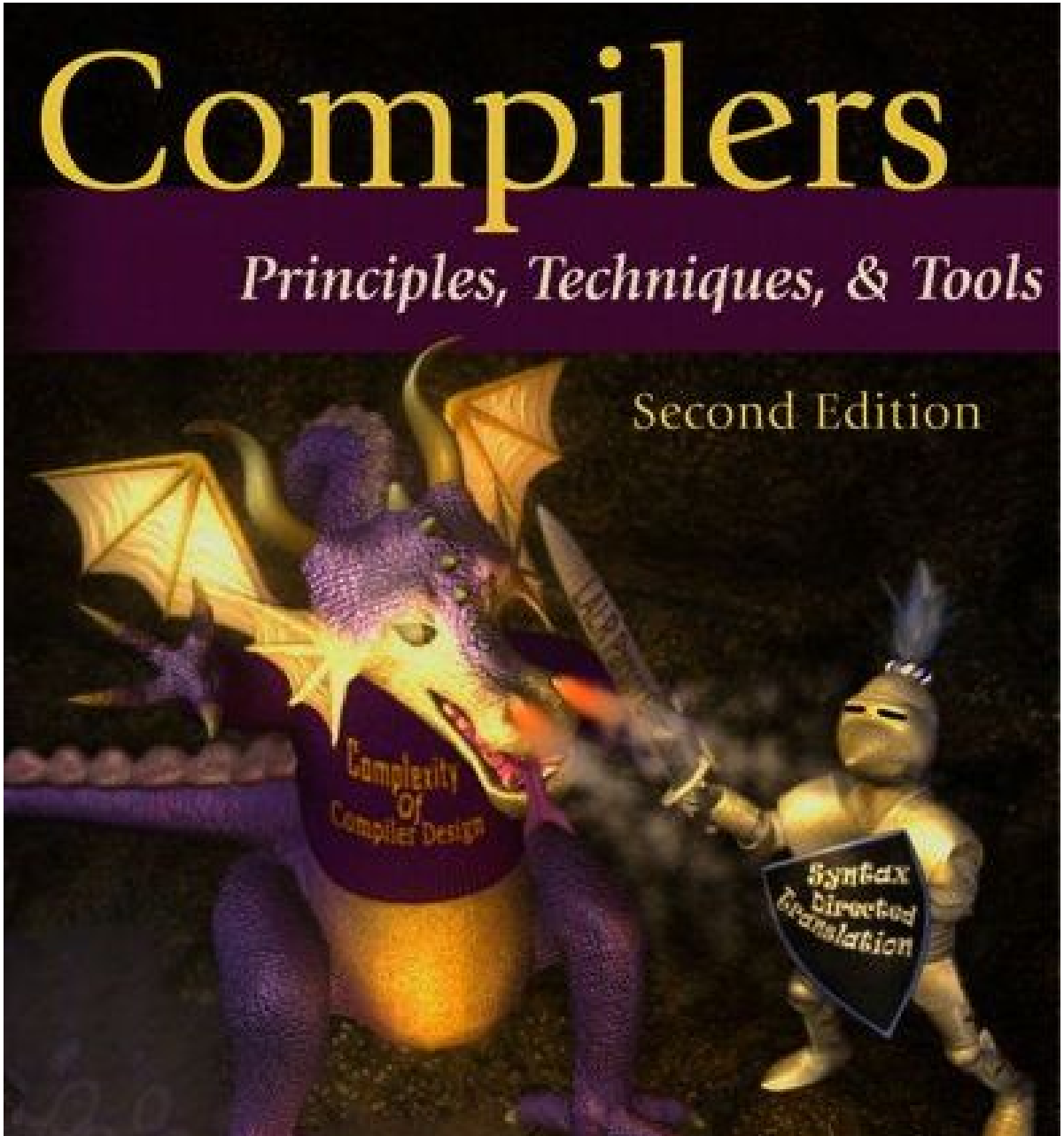
From Source to Machine Code

- A machine language consists of very simple commands.
- Writing a program in this language is a very tedious and error-prone.
- It is much easier to use instead high-level programming language.
- But, before a program can be run, it first must be translated by a compiler.

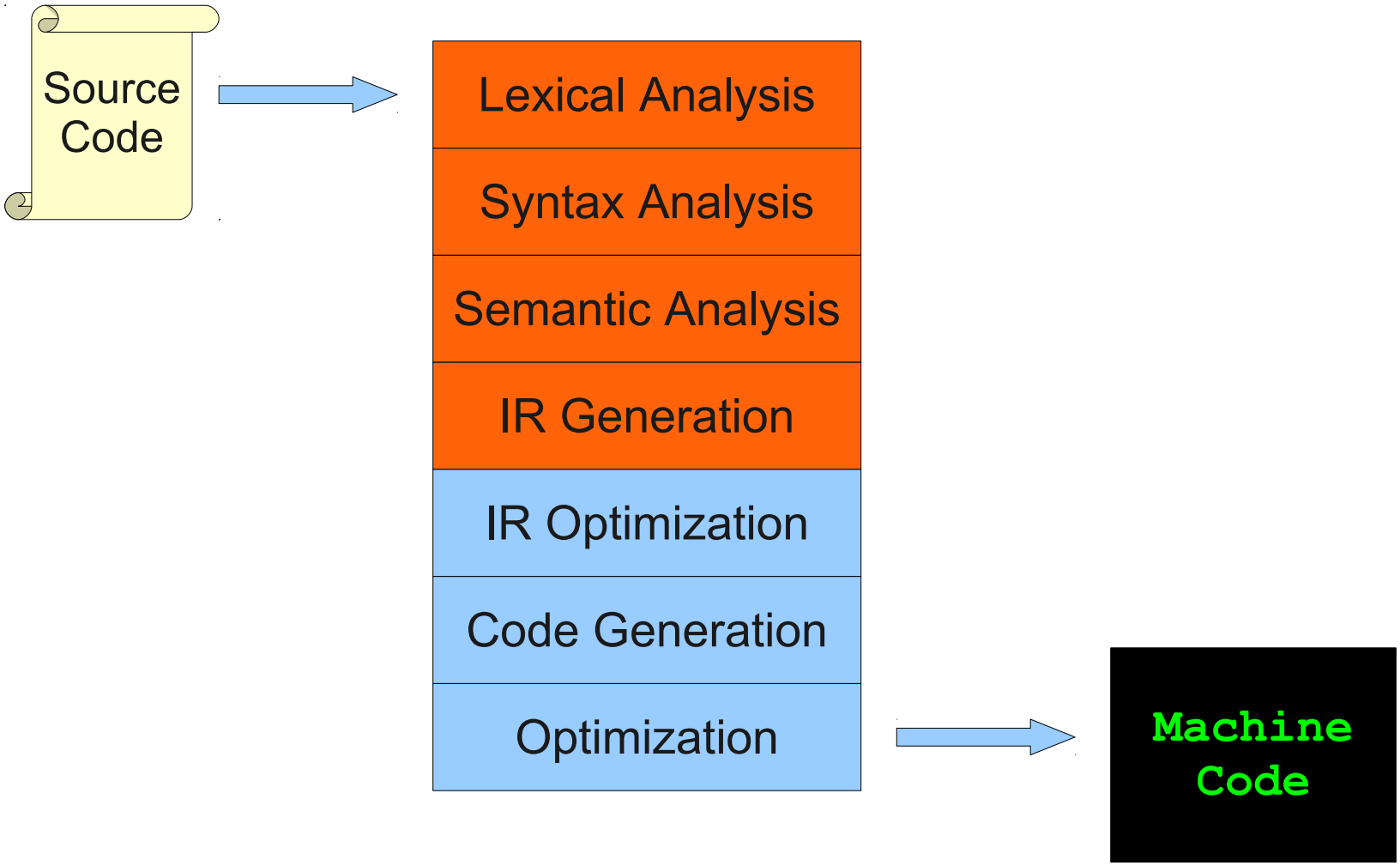
Advantages:

- notation is more intuitive and closer to human language
- compilers can spot some obvious programming mistakes
- programs are shorter than their equivalent written in machine language, and
- the same high-level program can be run on different machines, previous translation/compiling.

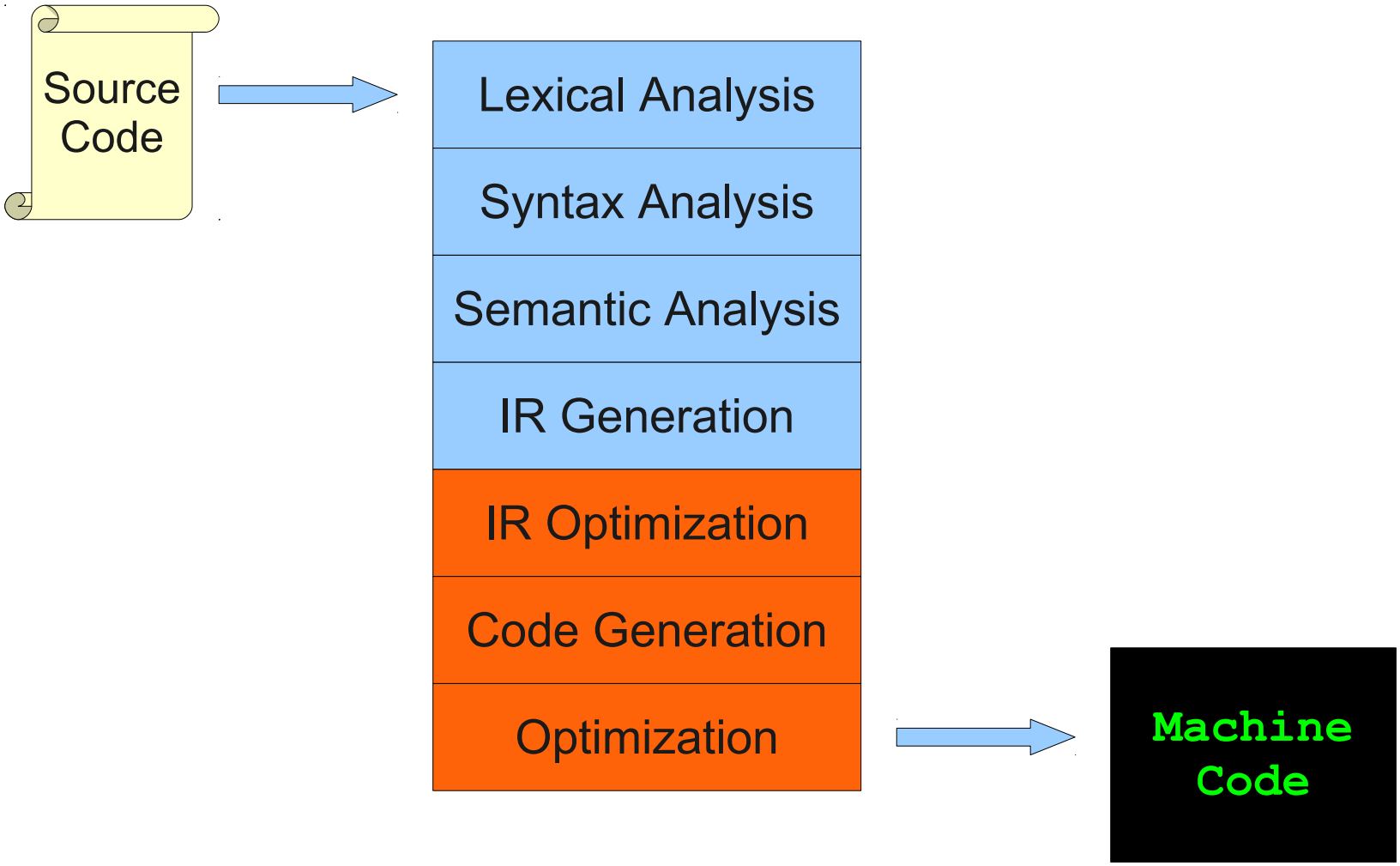
From Source to Machine Code



The Structure of a Modern Compiler

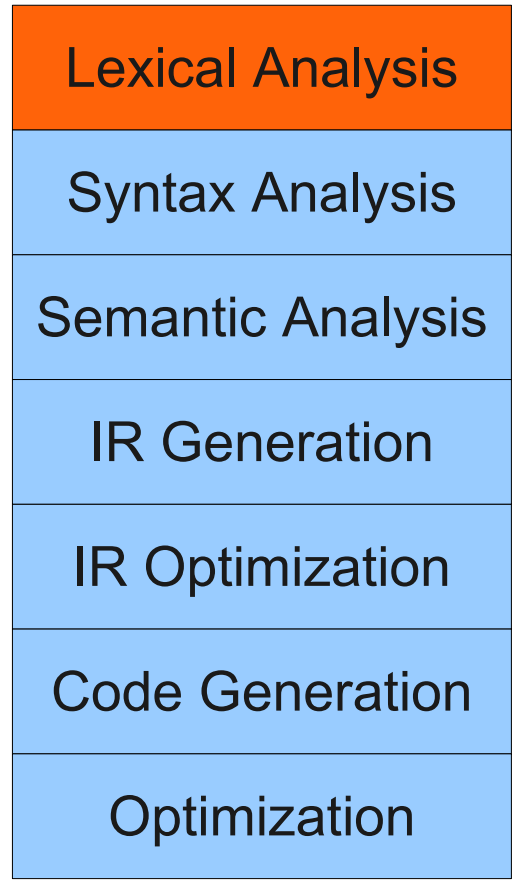


The Structure of a Modern Compiler



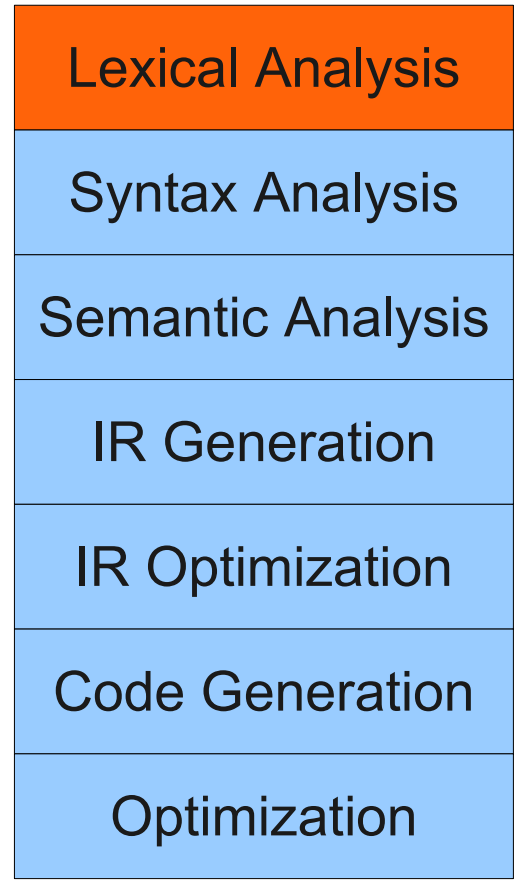
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

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



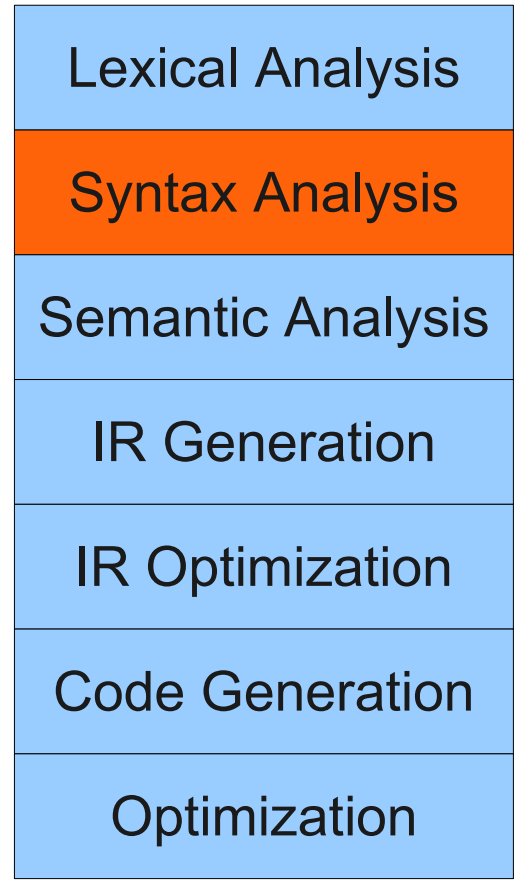
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

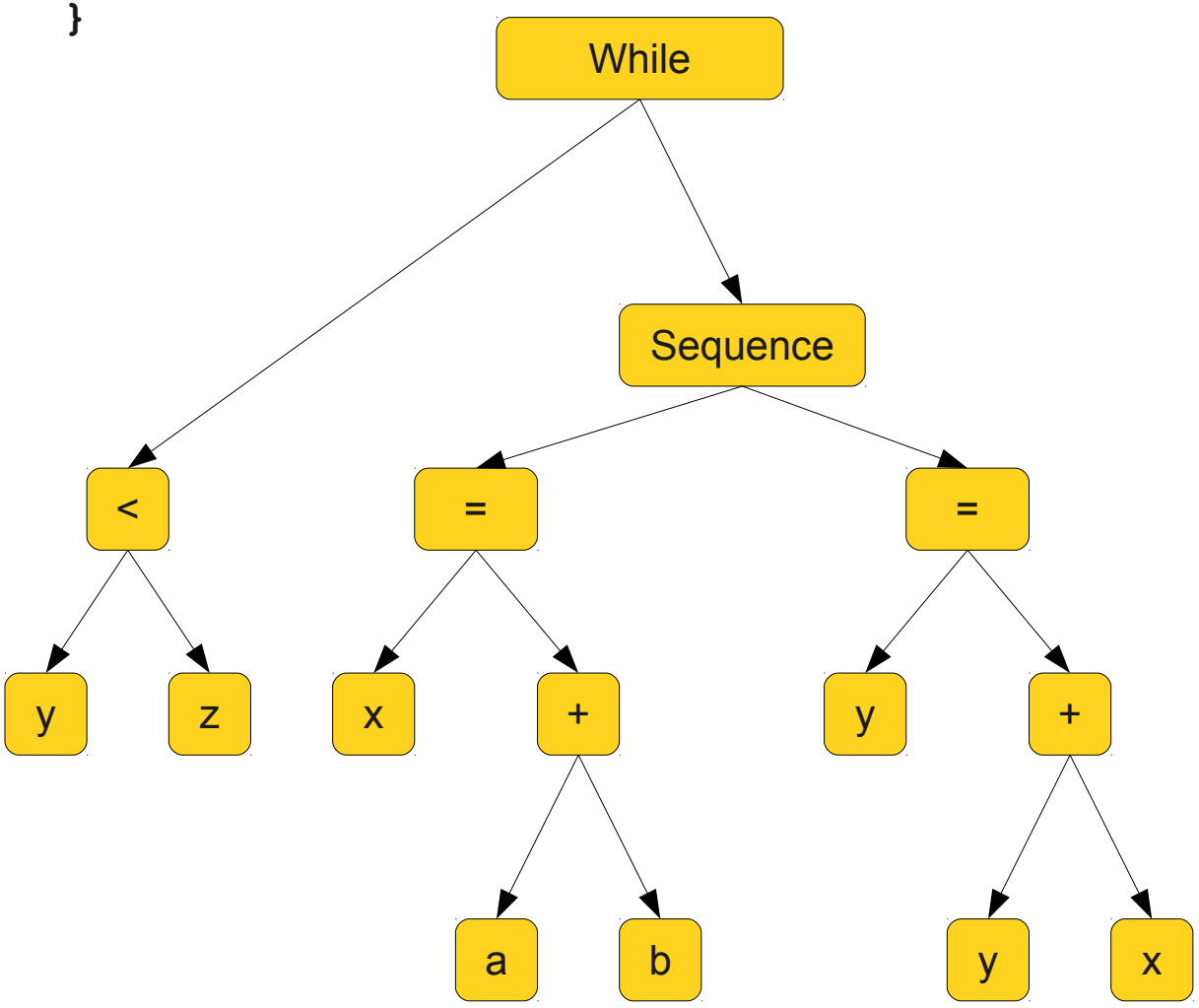


```
while (y < z) {
    int x = a + b;
    y += x;
}
```

```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Int
T_Identifier x
T_Assign
T_Identifier a
T_Plus
T_Identifier b
T_Semicolon
T_Identifier y
T_PlusAssign
T_Identifier x
T_Semicolon
T_CloseBrace
```

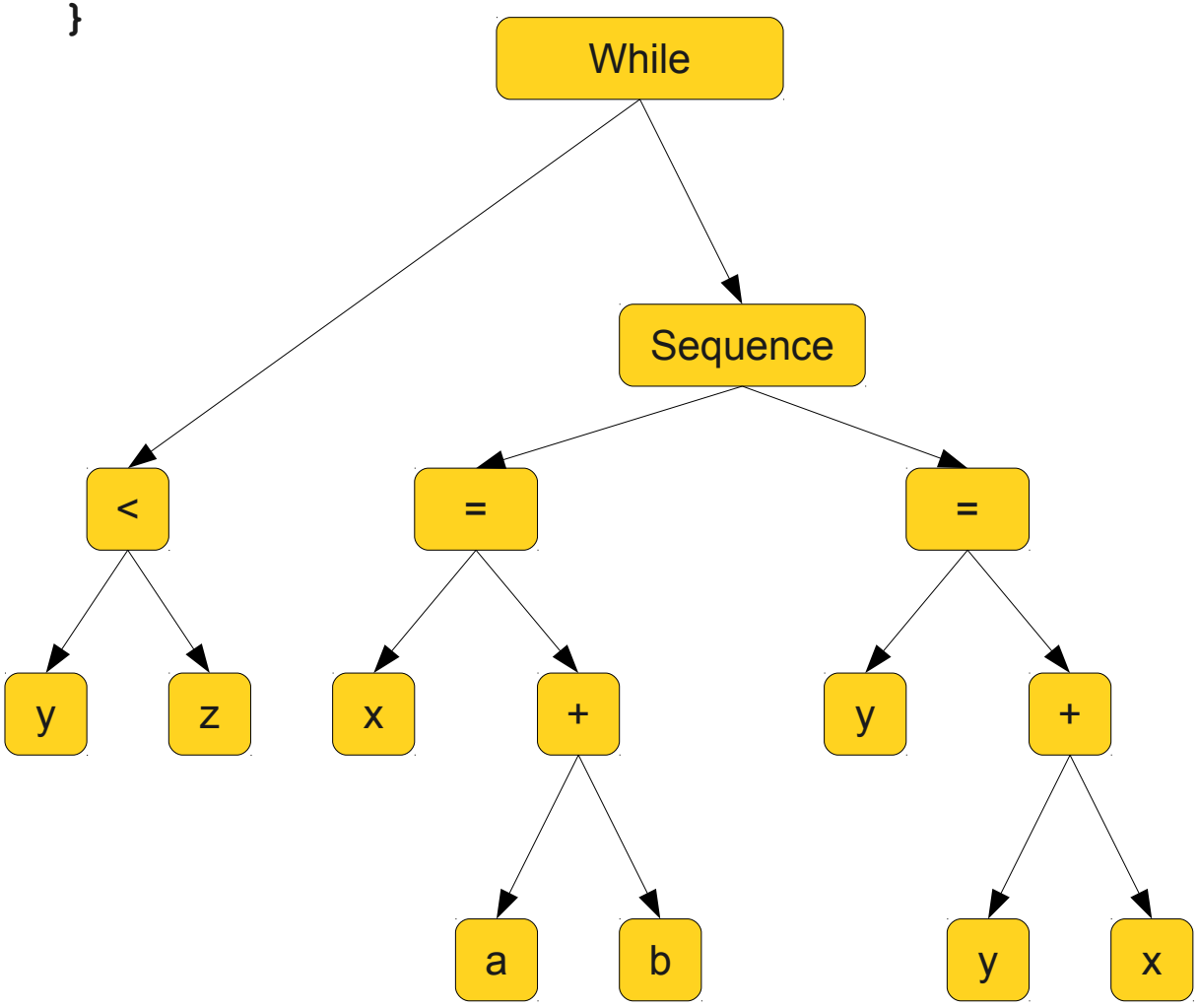


```
while (y < z) {
    int x = a + b;
    y += x;
}
```



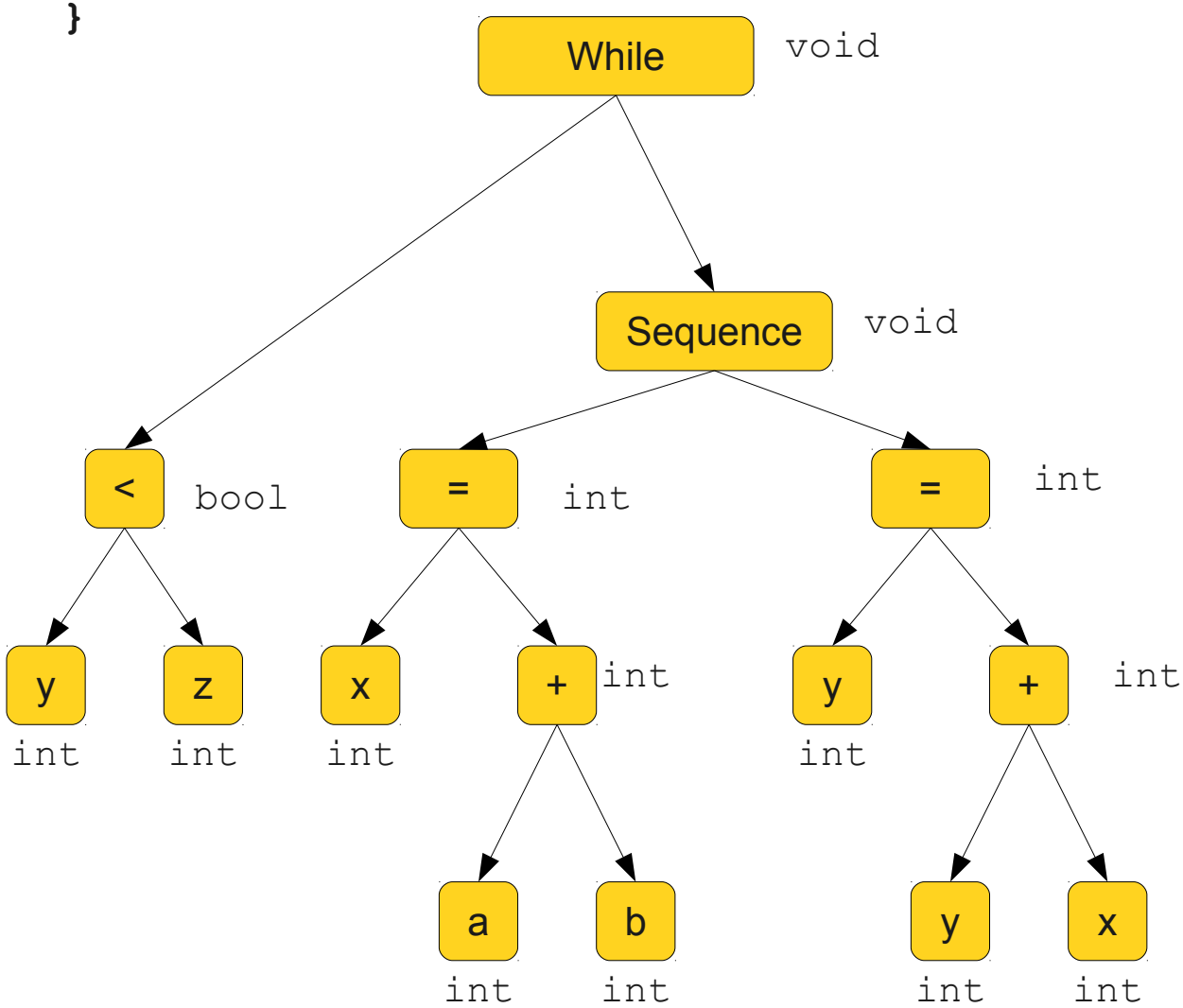
- Lexical Analysis
- **Syntax Analysis**
- Semantic Analysis
- IR Generation
- IR Optimization
- Code Generation
- Optimization

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



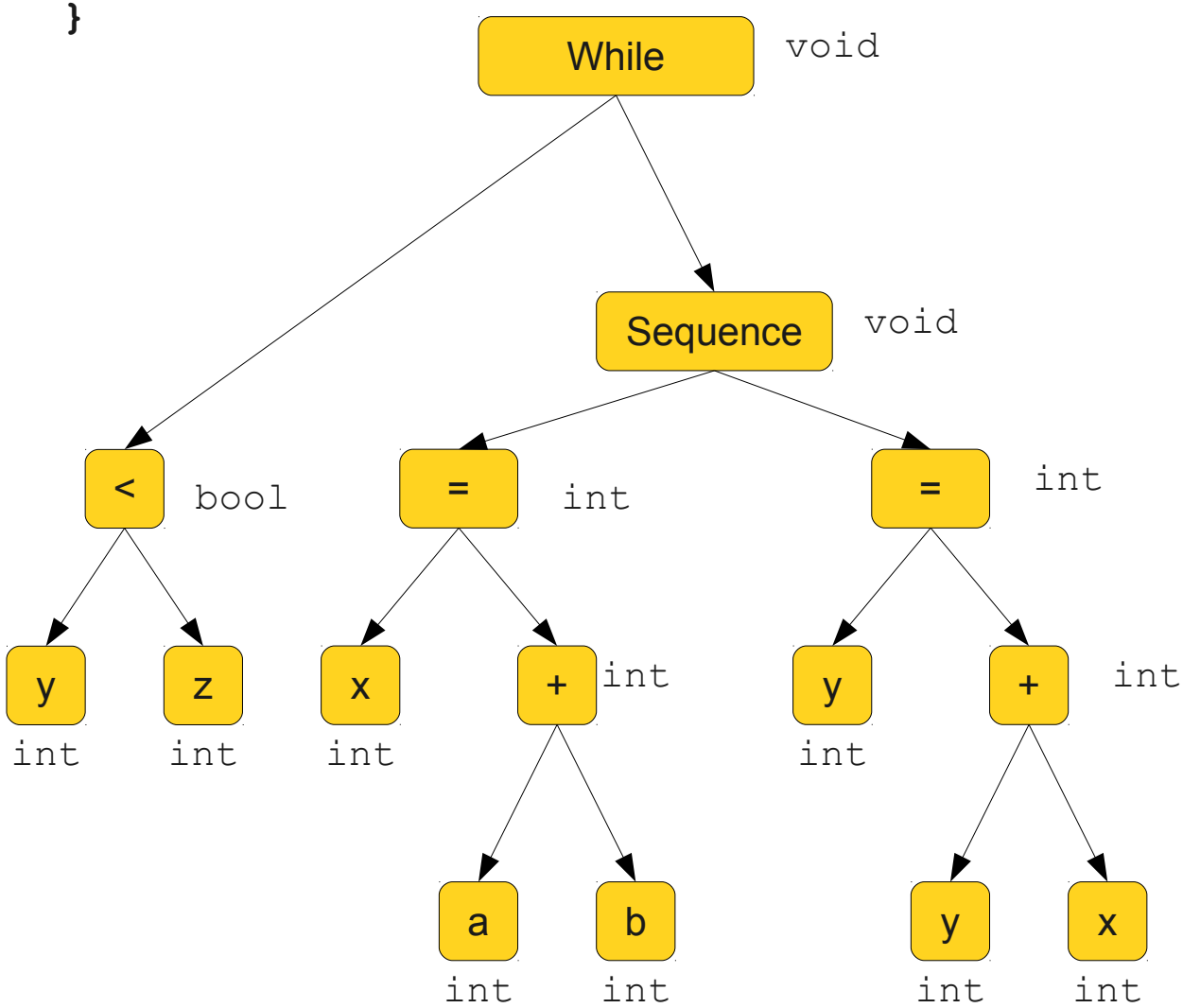
- Lexical Analysis
- Syntax Analysis
- **Semantic Analysis**
- IR Generation
- IR Optimization
- Code Generation
- Optimization

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



- Lexical Analysis
- Syntax Analysis
- **Semantic Analysis**
- IR Generation
- IR Optimization
- Code Generation
- Optimization

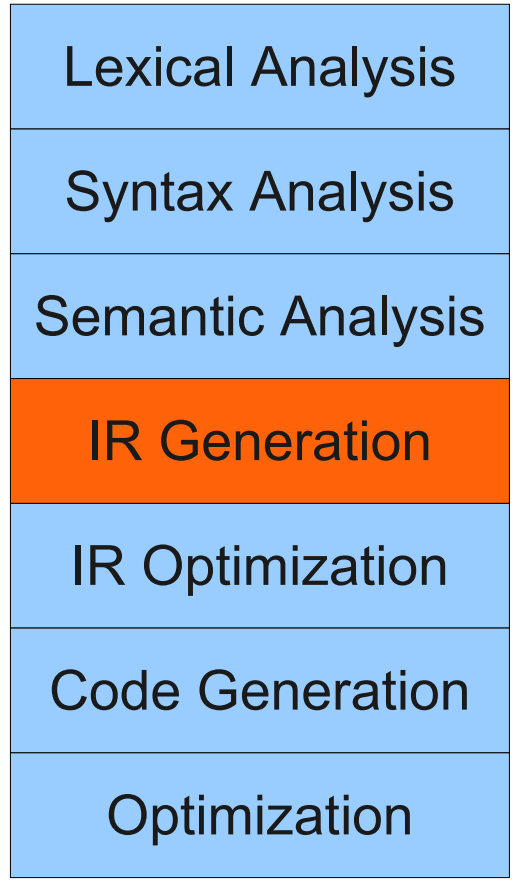
```
while (y < z) {
    int x = a + b;
    y += x;
}
```



- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- **IR Generation**
- IR Optimization
- Code Generation
- Optimization

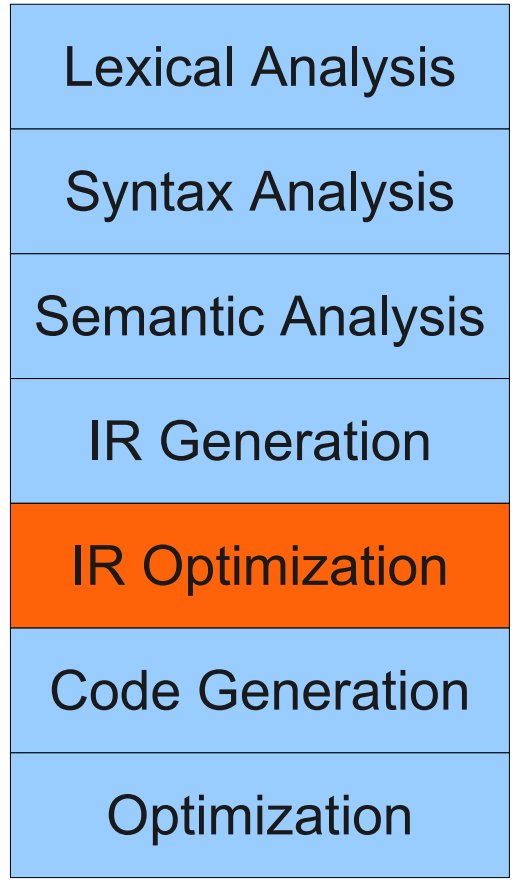
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

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



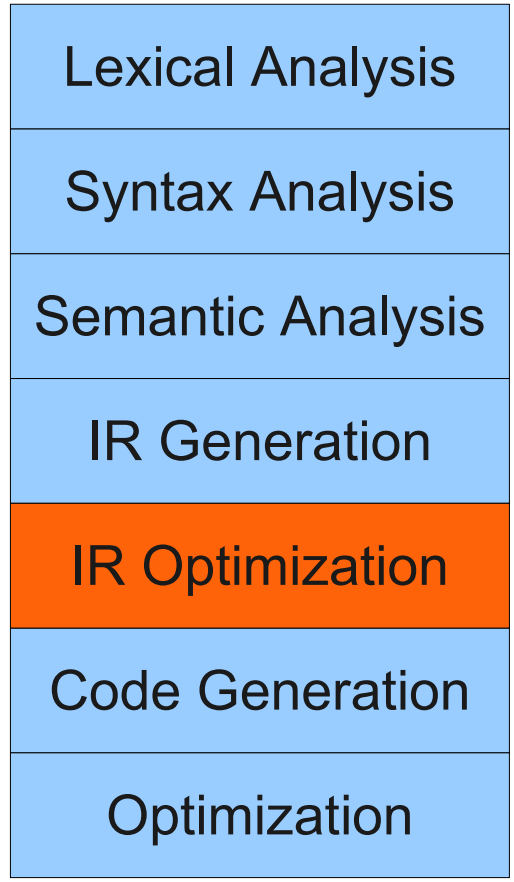
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

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



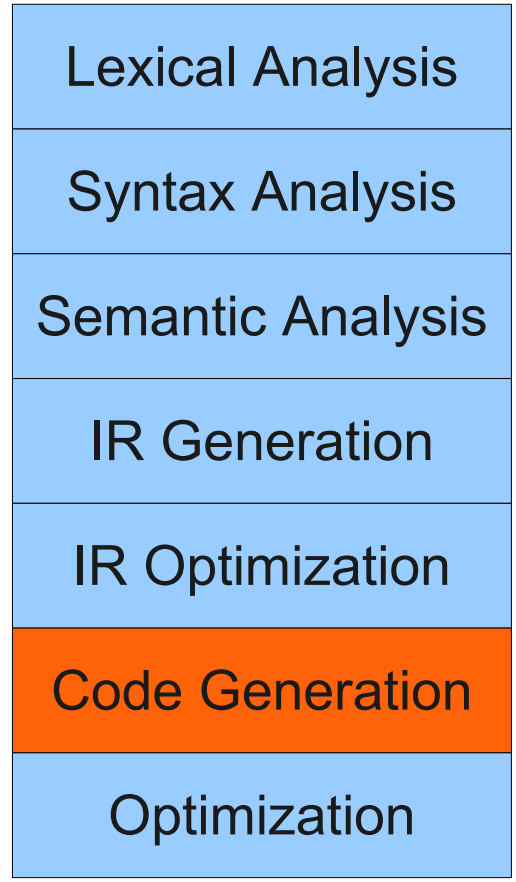
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

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



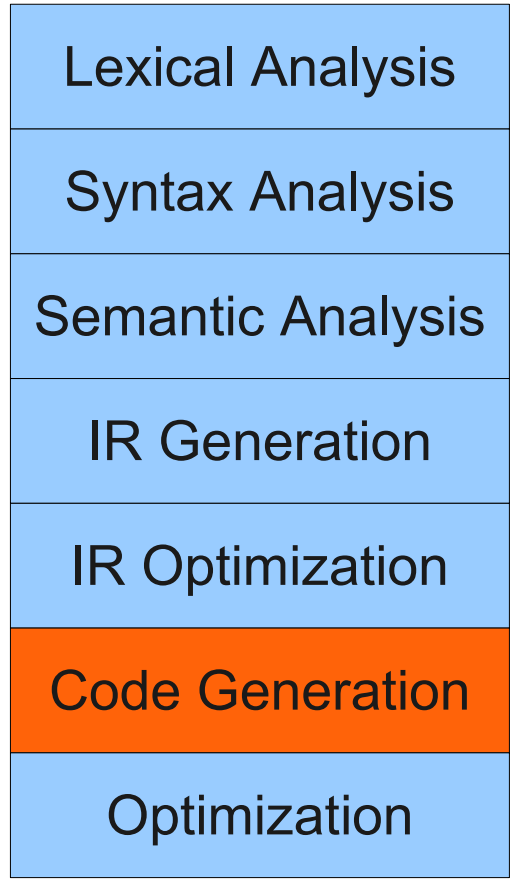
```
while (y < z) {
    int x = a + b;
    y += x;
}
```

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



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

```
        add $1, $2, $3
Loop:   add $4, $1, $4
        slt $6, $1, $5
        beq $6, loop
```



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

```
        add $1, $2, $3
Loop:   add $4, $1, $4
        slt $6, $1, $5
        beq $6, loop
```

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

```
        add $1, $2, $3
Loop:   add $4, $1, $4
        blt $1, $5, loop
```

Compiler Phases

Pseudo-SML Code

```
val result =
  let val x = 10 :: 20 :: 0x30 :: []
  in  List.map (fn a => 2 * 2 * a) x
  end
```

Discuss:

- what does the code do?
- write an equivalent C or Java code
- what would be the necessary steps to translate the source code to machine code?

Translation from Source to C to Machine Code

Equivalent C Code

- initialize the array
- map translated to a loop
- possible optimization:
  2*2=4 at compile time,
  result reuses the space of x.

```
int x[3] = {10, 20, 0x30};
int i = 0;
for (int i = 0; i < 3; i++) {
  x[i] = 4*x[i];
}
```

Translation from Source to C to Machine Code

Equivalent C Code

- initialize the array
- map translated to a loop
- possible optimization:
  2*2=4 at compile time,
  result reuses the space of x.

```
int x[3] = {10, 20, 0x30};
int i = 0;
for (int i = 0; i < 3; i++) {
  x[i] = 4*x[i];
}
```

Pseudo MIPS assembly code

- three-address-like code
- registers instead of variables
- explicit load and store ops
- shift instead of multiplication

```
load_imm  $2, 0      # $2 counter
load_addr $3, x_addr # $3 address of x
load_imm  $5, 3      # $5 stores ct 3
loop:
    branch_ge  $2, $5, end
    load_word  $4, 0($3) # $4 holds x[i]
    shift_left $4, $4, 2 # multiply by 4
    store_word $4, 0($3) # store in x[i]
    add_imm    $3, $3, 4 # next x addr
    add_imm    $2, $2, 4 # i = i + 1
    jmp        loop
end:
```

Compiler Phases

- Front-End:
  - Lexical Analysis: split program to individual words that make sense: *My mother coooookes dinner not.*
  - Syntactical Analysis: does the composition of words respect the language grammar? *My mother cooks dinner not*.
  - Type Checking: are operators/functions applied to variables of correct types? *Adding 1 cat with 1 dog results in 2 penguins.*

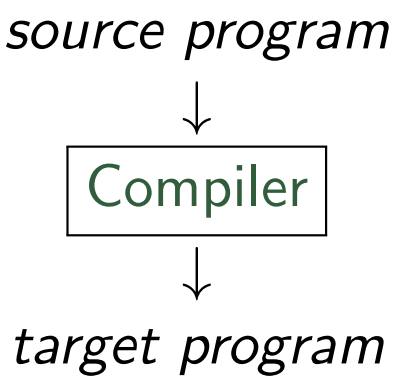
Compiler Phases

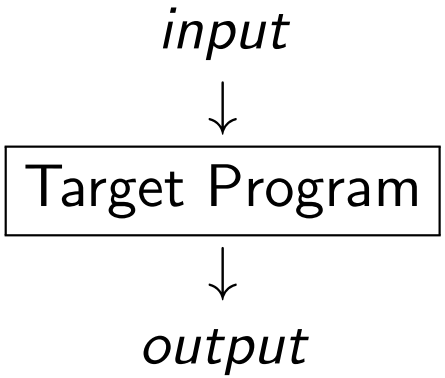
- Front-End:
  - Lexical Analysis: split program to individual words that make sense: *My mother coooookes dinner not.*
  - Syntactical Analysis: does the composition of words respect the language grammar? *My mother cooks dinner not*.
  - Type Checking: are operators/functions applied to variables of correct types? *Adding 1 cat with 1 dog results in 2 penguins.*
- Intermediate-Language (IL) Code Generation & Optimizations:
  - IL and optimizations reused across several source languages.
  - Moving towards lower-level representations, e.g., how are arrays represented in memory

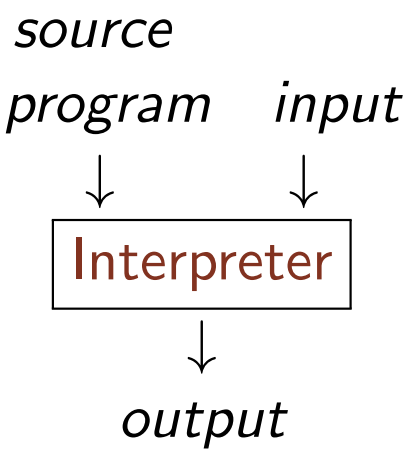
Compiler Phases

- Front-End:
  - Lexical Analysis: split program to individual words that make sense: *My mother coooookes dinner not.*
  - Syntactical Analysis: does the composition of words respect the language grammar? *My mother cooks dinner not*.
  - Type Checking: are operators/functions applied to variables of correct types? *Adding 1 cat with 1 dog results in 2 penguins.*
- Intermediate-Language (IL) Code Generation & Optimizations:
  - IL and optimizations reused across several source languages.
  - Moving towards lower-level representations, e.g., how are arrays represented in memory
- Back-End: Generation of Machine-Code (MC)
  - How to translate IL constructs to MC, e.g., function calls?
  - What sequence of machine instrs best implements the IL code?
  - Machine resources are limited, e.g., arbitrary number of code variables need to be mapped to a finite number of registers.

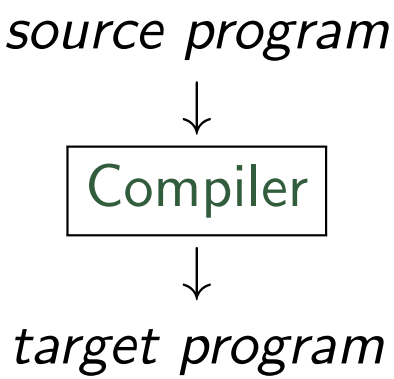
Compilation vs. Interpretation

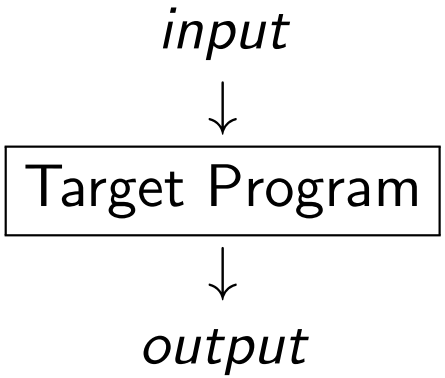


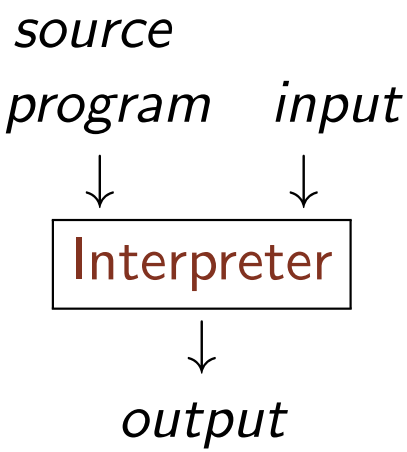




Compilation vs. Interpretation

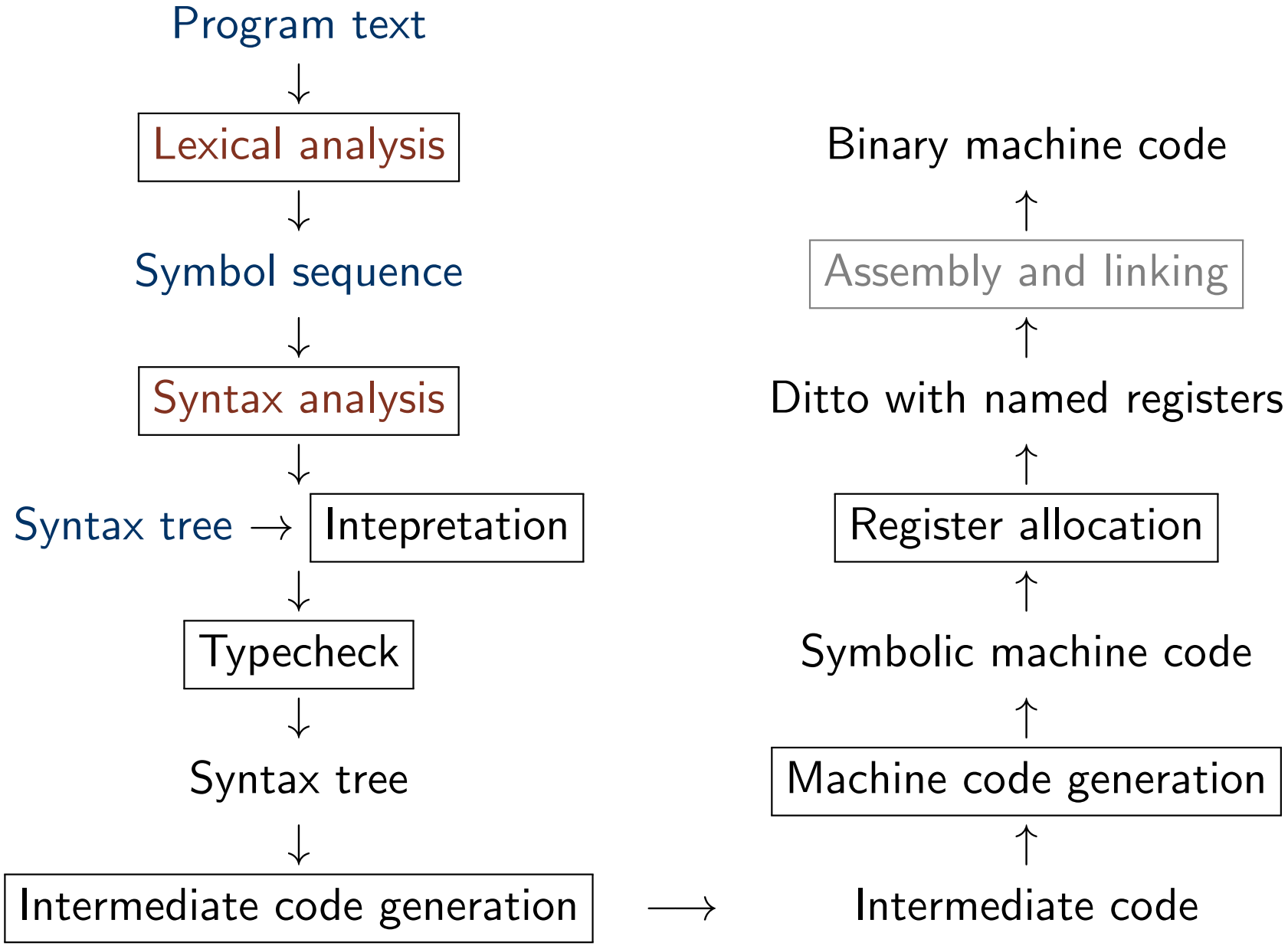






- Compilation results in a lower-level language program, e.g., machine code, which can be run on various inputs.
- Interpretation directly *executes* one by one the operations specified in the *source program* on the *input* supplied by the user, by using the facilities of its implementation language.

Structure of a Compiler



Another Example

Objective: to translate programs such as the following simple one

```
{
      int i; int j; float[100] a; float v; float x;

      while ( true ) {
            do i = i+1; while ( a[i] < v );
            do j = j-1; while ( a[j] > v );
            if ( i >= j ) break;
            x = a[i]; a[i] = a[j]; a[j] = x;
      }
}
```

Figure 2.1: A code fragment to be translated

Another Example (ctd.)

The compiler front end translates the program into the form:

```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

Figure 2.2: Simplified intermediate code for the program fragment in Fig. 2.1