# Programming Language Implementation – Introduction

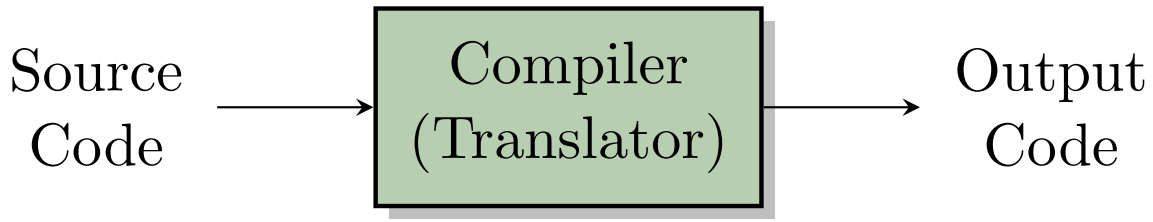
Sujit Kumar Chakrabarti

IIITB

IIIT-B

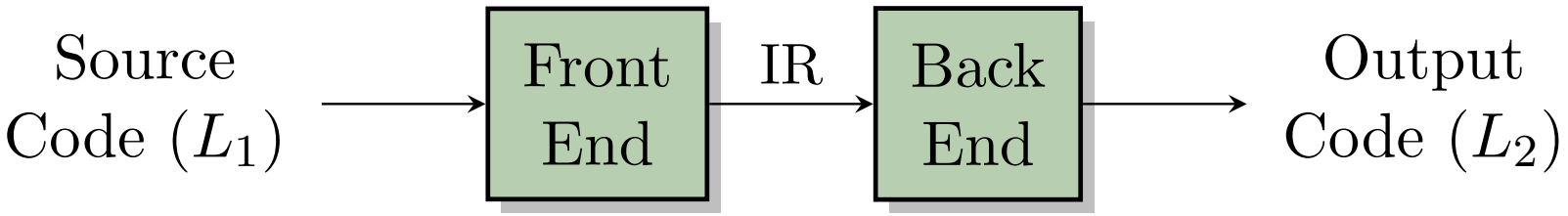
# Compilers

## Overview



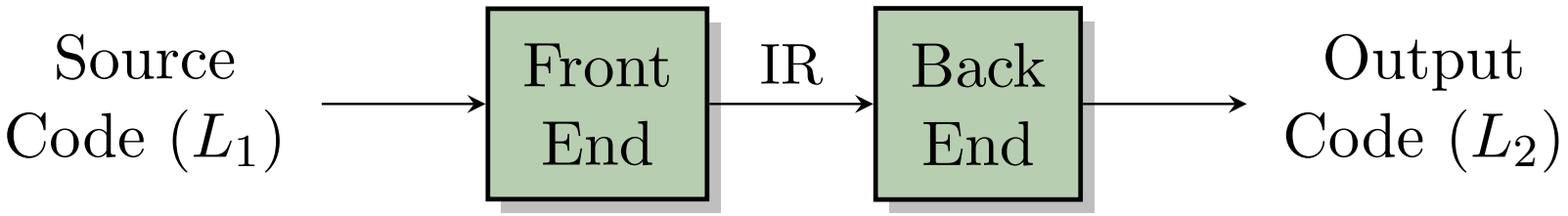

- Translates input program to output program
- Input program (source code ) in high-level language (e.g. C, Java, Python etc.)
- Output program (target code) in low-level language (e.g. machine code, assembly, byte-code etc.)
- Source code $\equiv$ Target code

# Compiler Structure




- Two functions:
  1. Read source code and analyse for errors
  2. Translate into the target code
- Two parts:
  1. Front end
  2. Back end

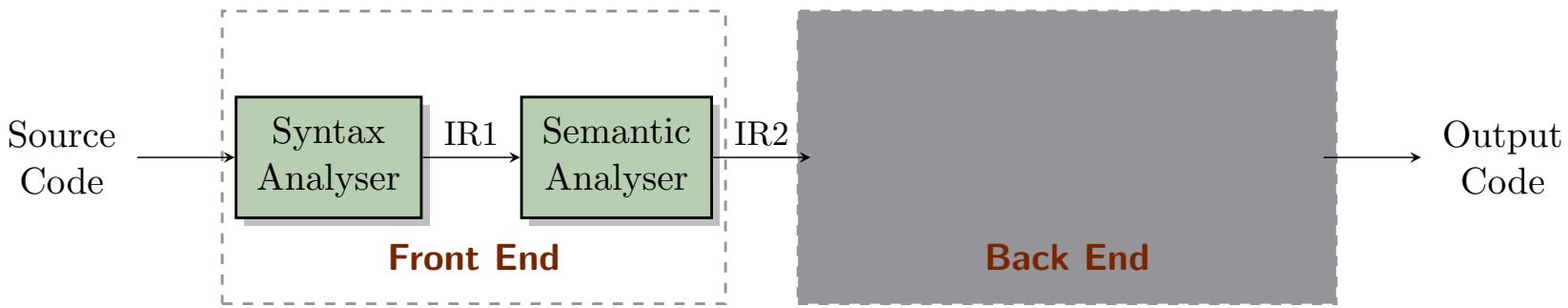
# Compiler Structure




- Two functions:
  1. Read source code and analyse for errors
  2. Translate into the target code
- Two parts:
  1. Front end
  2. Back end
- **Intermediate representation (IR)**
  - A data structure output from front end and input to the backend
  - Several phases; several IRs

# Compiler Structure

## Front End



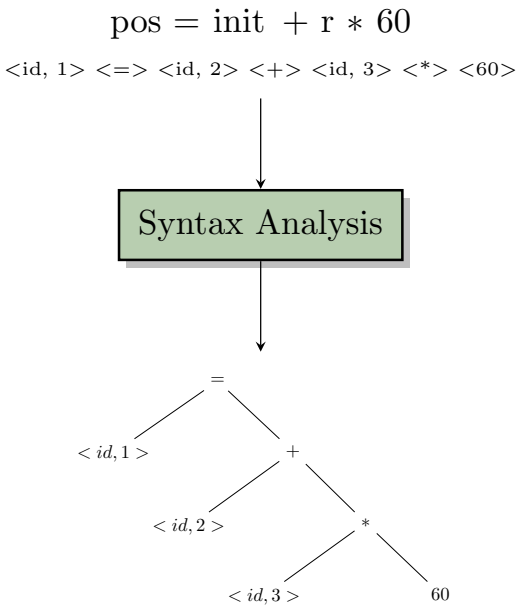
# Compiler Structure

## Syntax Analysis

- **Syntax errors:**
  1. Example 1: `for i in range(10)`
  2. Example 2: `void f(x) { ... }`
- When program doesn't follow grammatical constructs of the input PL
- **Well formed programs:** Programs with *no* syntax errors
- **Ill formed programs:** Programs *with* syntax errors

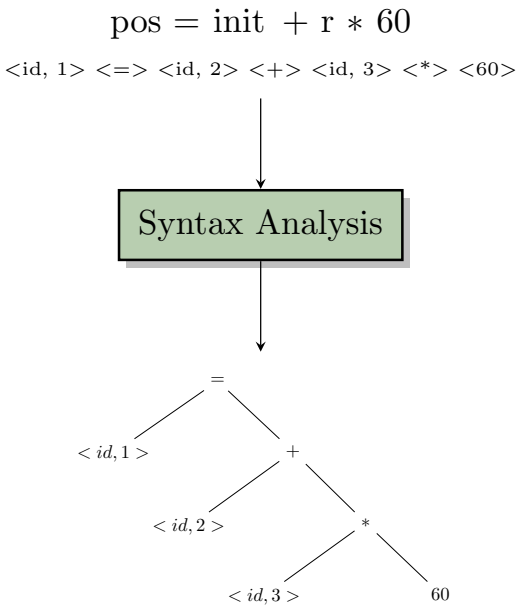
# Compiler Structure

## Syntax Analysis



# Compiler Structure

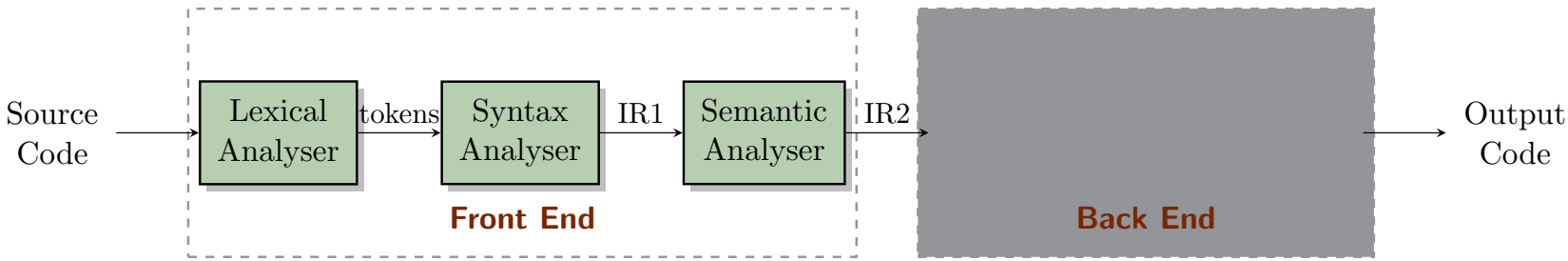
## Syntax Analysis




- Successful syntax analysis $\longrightarrow$ Abstract syntax tree (AST)
- More details later

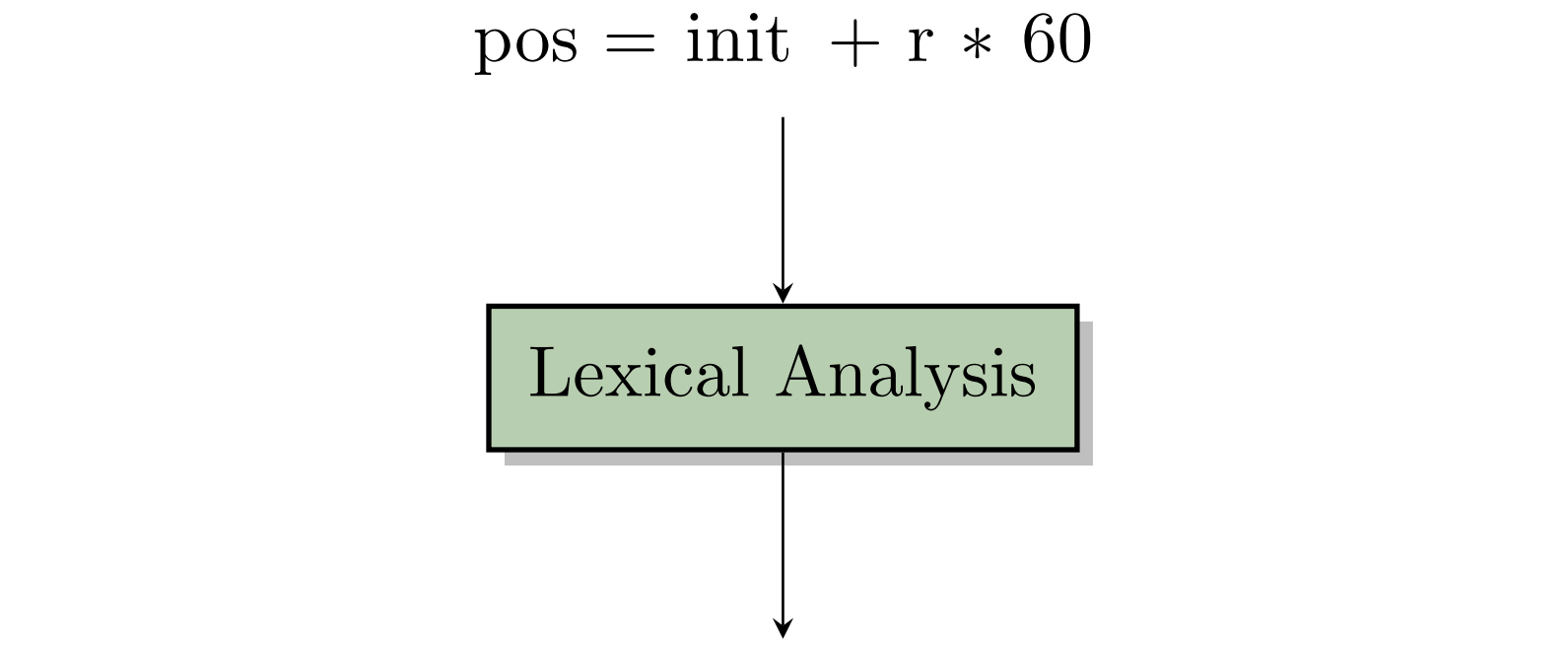
# Compiler Structure

## Lexical Analyser



# Compiler Structure

## Lexical Analysis



# Compiler Structure

## Semantic Analysis – Functions

1. Variable binding:

```
let x = 20 in
let sum1 = 2 * x in
let x = 100 and z = 5 in
  sum1 - z + x
```

2. Type checking:

```
void add(int x, int y) { return x + y; }
add(1, 2);
```

3. Type checking:

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
```

# Compiler Structure

## Semantic Analysis – Semantic errors

- **Semantic errors:** Error associated with the meaning of the program
- Natural language example:

  *Table goes to school.*

# Compiler Structure

## Semantic Analysis – Semantic errors

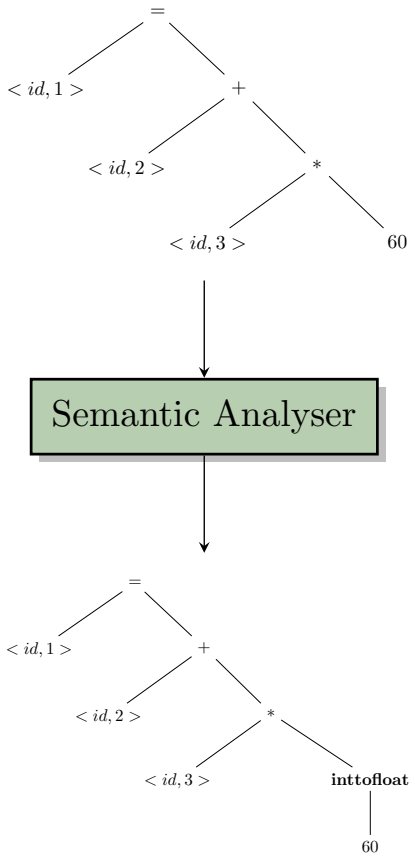
- **Semantic errors:** Error associated with the meaning of the program
- Natural language example:

  *Table goes to school.*

- **Ill-typed programs:** Well formed programs that don't make sense.
- Examples:
  1. Example 1: `String s = 10;`
  2. Example 2: `void f() \{ return x; \}`

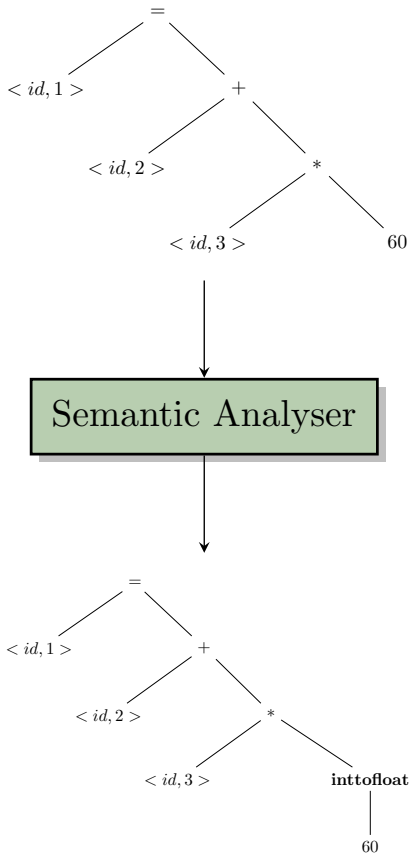
# Compiler Structure

## Semantic Analysis



# Compiler Structure

## Semantic Analysis



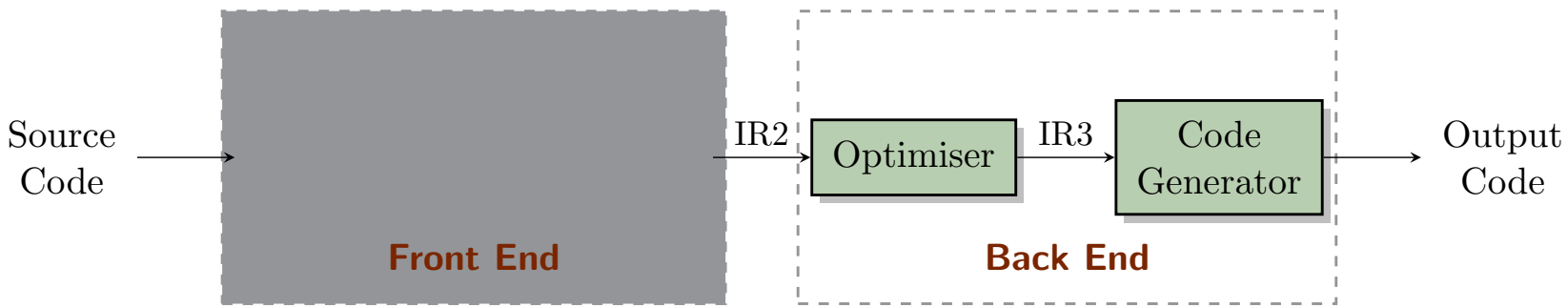
# Compiler Structure

## Next Module

Compiler Backend

# Compiler Structure

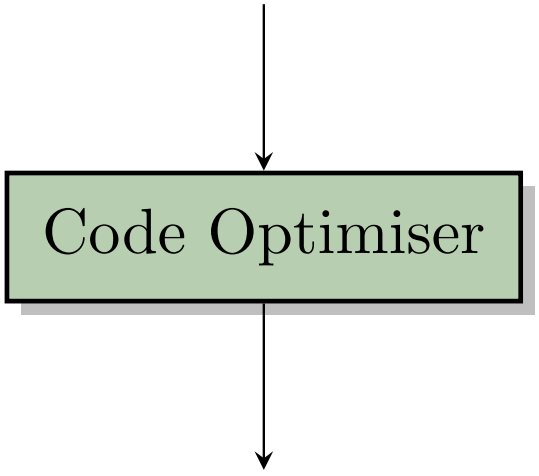
## Back End



# Compiler Structure

## Code Optimisation

$$t1 = \text{inttofloat}(60)$$
$$t2 = id3 * t1$$
$$t3 = id2 + t2$$
$$id1 = t3$$




$$t1 = id3 * 60.0$$
$$id1 = id2 + t1$$

# Compiler Structure

## Code Optimisation

### Example – Loop optimisation

```
for(int i = 0; i < N; i++) {
  t = a + b;
  sum += t + i;
}
```

$\Rightarrow$

```
t = a + b;
for(int i = 0; i < N; i++) {
  sum += t + i;
}
```

# Compiler Structure

## Code Optimisation

- Numerous algorithms
- Local, global, interprocedural
- Control flow analysis, data flow analysis
- Machine independent and machine dependent
- ...

# Compiler Structure

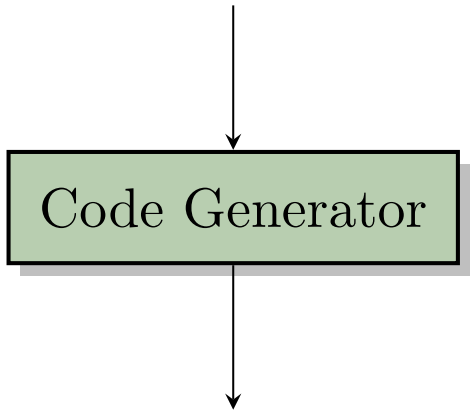
## Code Optimisation

- Numerous algorithms
- Local, global, interprocedural
- Control flow analysis, data flow analysis
- Machine independent and machine dependent
- ...
- Largest and most complex module of modern compilers
- Active area of research

# Compiler Structure

## Code Generation

$t1 = id3 * 60.0$
$id1 = id2 + t1$

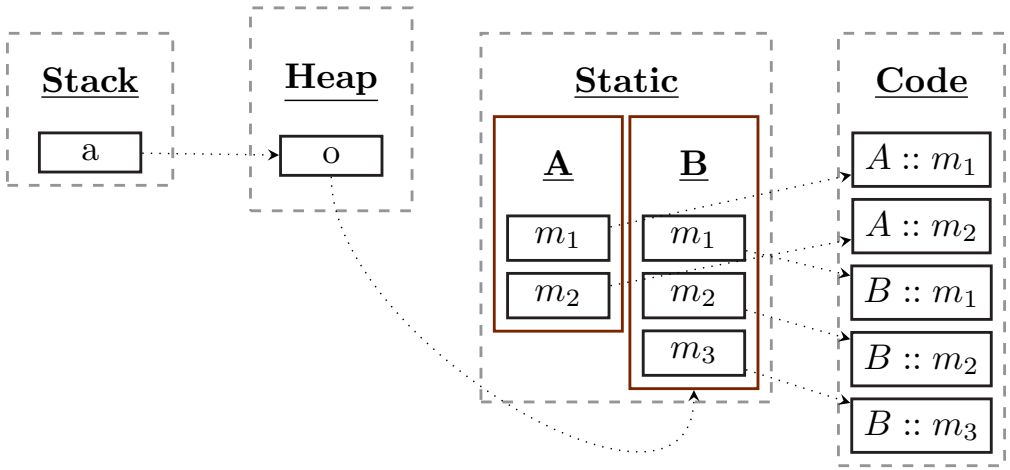



```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

# Compiler Structure

## Code Generation

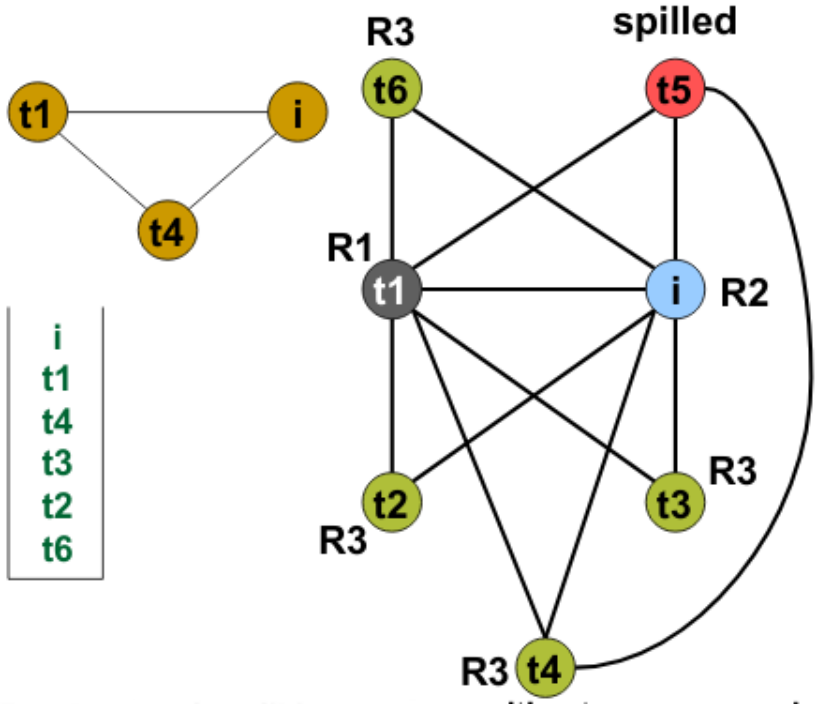
### Runtime Organisation




- Placement of various parts of the target code in the appropriate parts of the virtual memory
- Not everything generated statically
- Certain things have to be created by the target code at runtime.

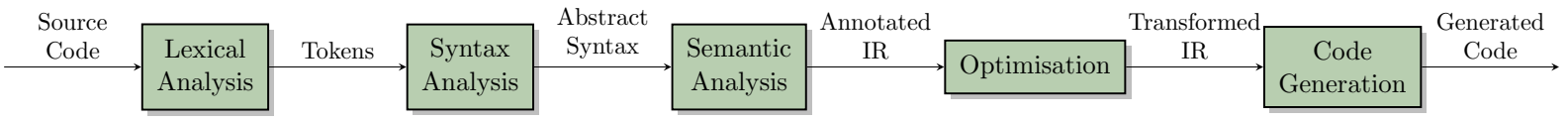
# Compiler Structure

## Register allocation




```
1.        R1 = 202
2.        R2 = 1
3.  L1:   R3 = i>100
4.        if R3 goto L2
5.        R1 = R1 - 2
6.        R3 = addr(a)
7.        R3 = R3 - 4
8.        t5 = 4*R2
9.        R3 = R3 + t5
10.      *R3 = R1
11.       R2 = R2+1
12.       goto L1
13. L2:
```

# Compiler Structure



# Summary – PLDI

## Applications

- Hardware synthesis
- Database query interpreter
- IDEs
- Debuggers
- Bug finders – Lint, Valgrind

# Summary – PLDI

## Why study PLDI

- Vibrant area of study – new PLs everyday!
- Sophisticated theory + Practical system building
- Deeper knowledge of data-structures, algorithms, software design

# Summary – PLDI

## Why study PLDI

- Vibrant area of study – new PLs everyday!
- Sophisticated theory + Practical system building
- Deeper knowledge of data-structures, algorithms, software design