Working of Compilers

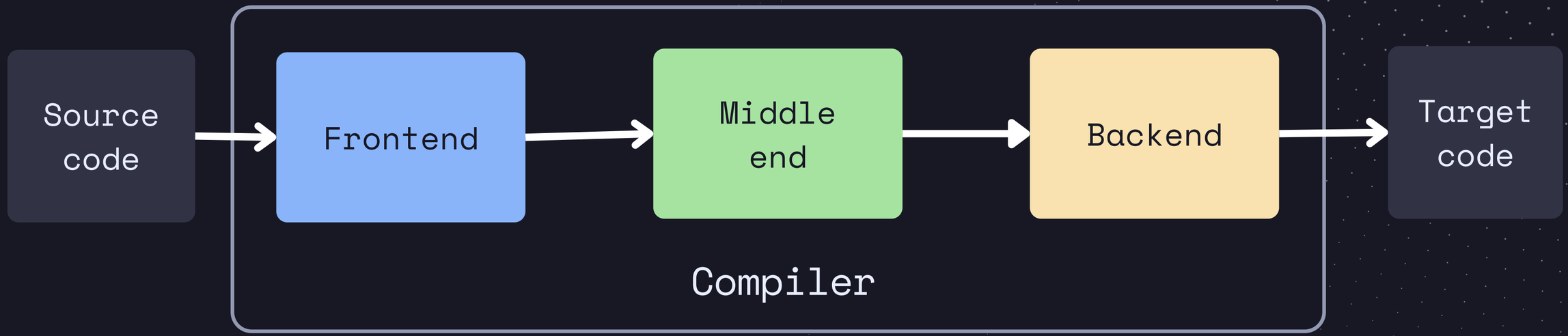
Shikhar Soni

What is a Compiler?

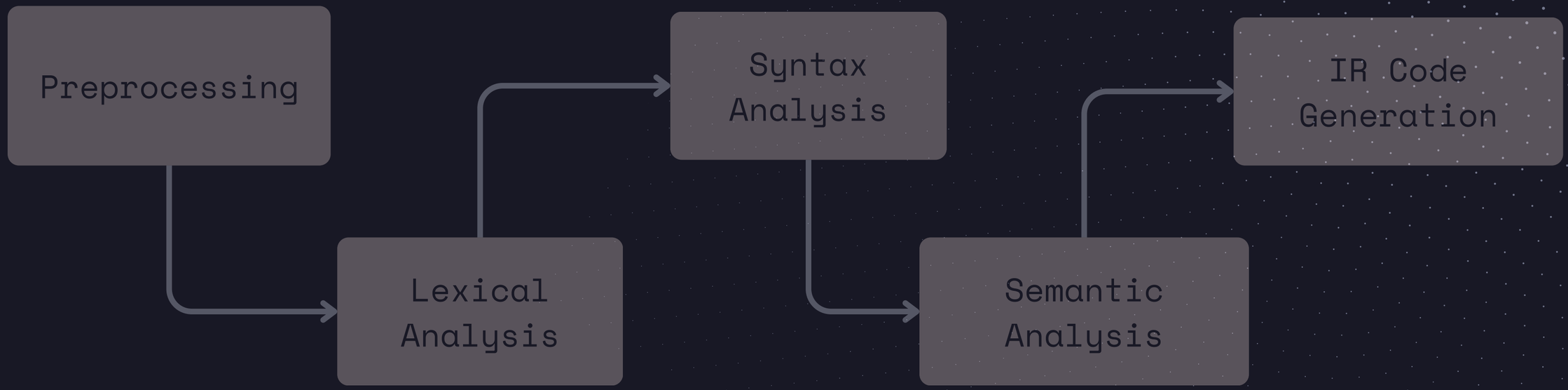
A program which translates code written in a high-level language(like C, Go) to a low-level language(like assembly, machine code).

Compiler Design

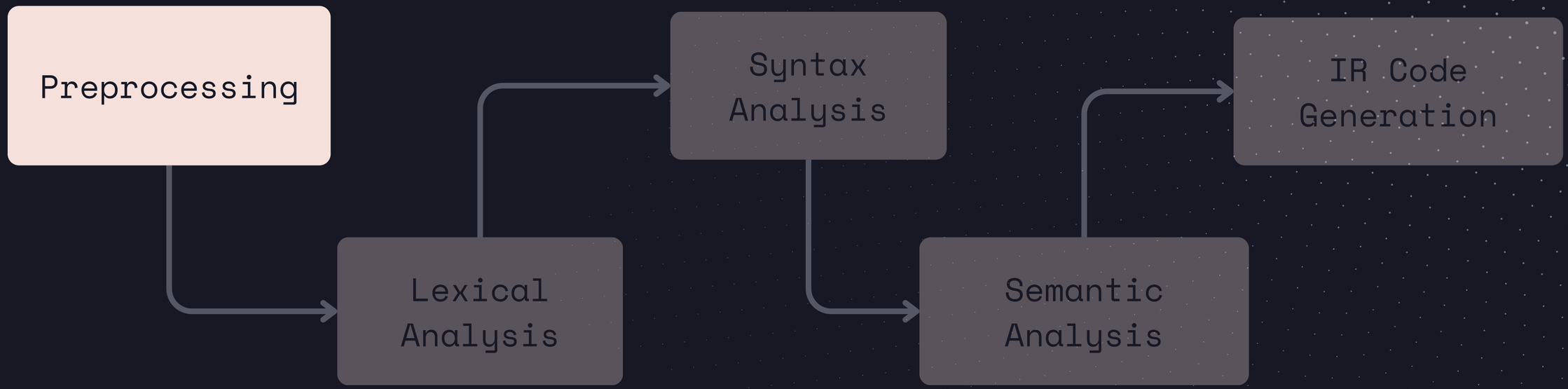
A compiler consists of three parts:



Front-end


Preprocessing
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Code Generation

Front-end


Preprocessing
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Code Generation

Front-end

```
$ cat batcall.c
#include <stdio.h>

int main(){
    printf("I am batman!\n");
    return 0;
}

$ clang-14 -E batcall.c -o preprocessed.i
$ wc -l preprocessed.i
757 preprocessed.i
```

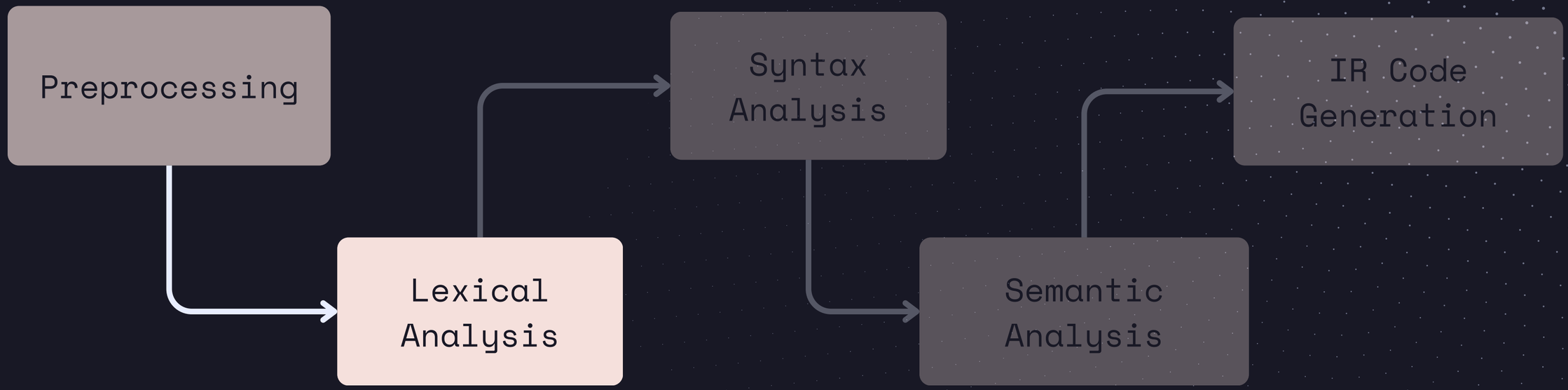
```
$ cat batcall.c
#include <stdio.h>

int main(){
    printf("I am batman!\n");
    return 0;
}

$ clang-14 -E batcall.c -o preprocessed.i
$ wc -l preprocessed.i
757 preprocessed.i
```

```
$ clang-14 preprocessed.i
$ ./a.out
I am batman!
```

Front-end

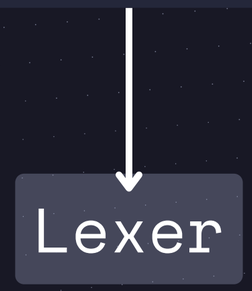

Preprocessing
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Code Generation

Front-end

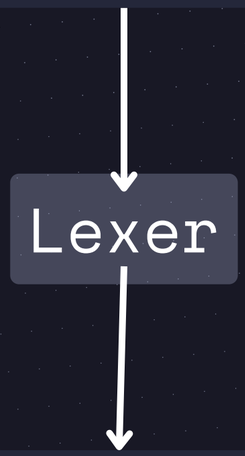
Front-end

```c
int a = 24;
if (a == 42) printf("You have found the answer to the universe\n");
```

```c
int a = 24;
if (a == 42) printf("You have found the answer to the universe\n");
```

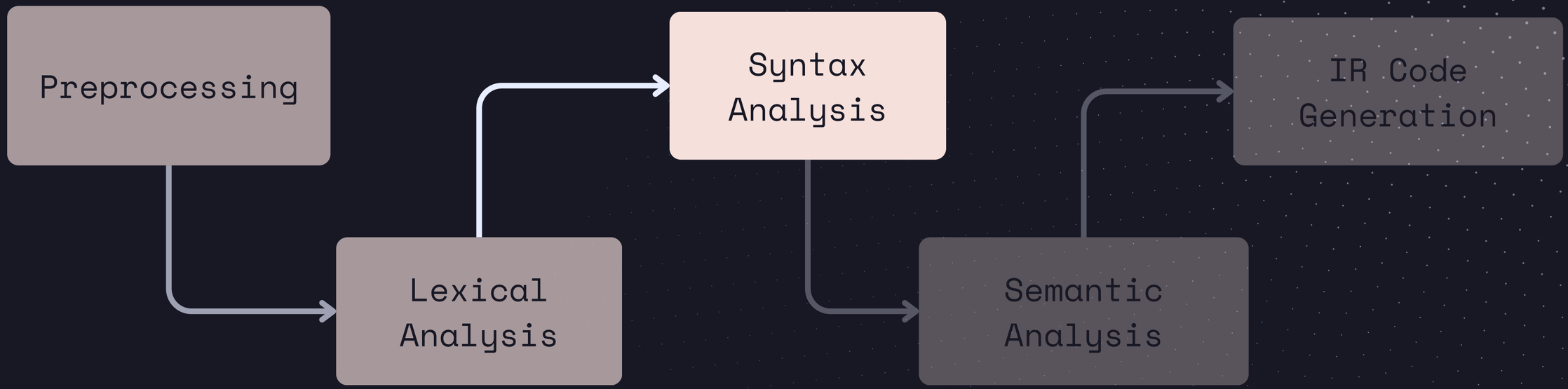


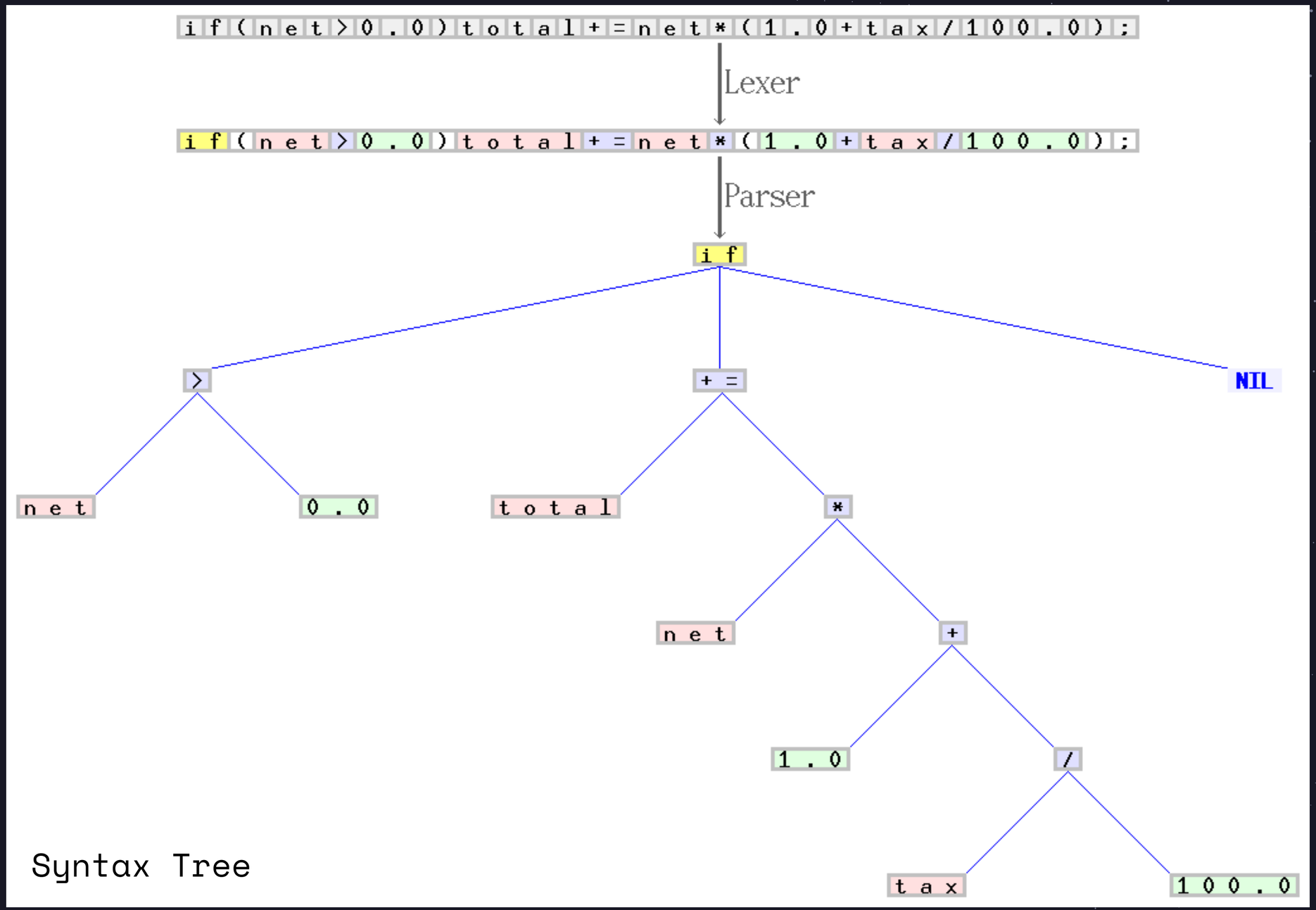
```c
int a = 24;
if (a == 42) printf("You have found the answer to the universe\n");
```




```
[(keyword, int), (identifier, a), (operator, =), (literal, 24)
(punctuator, ;), (keyword, if), (punctuator, (), (identifier, a),
(operator, ==), (literal, 42), (punctuator, )), (identifier, printf),
(punctuator, (), (literal, “You have found the answer to the
universe\n”), (punctuator, )), (punctuator, ;)]
```

Front-end

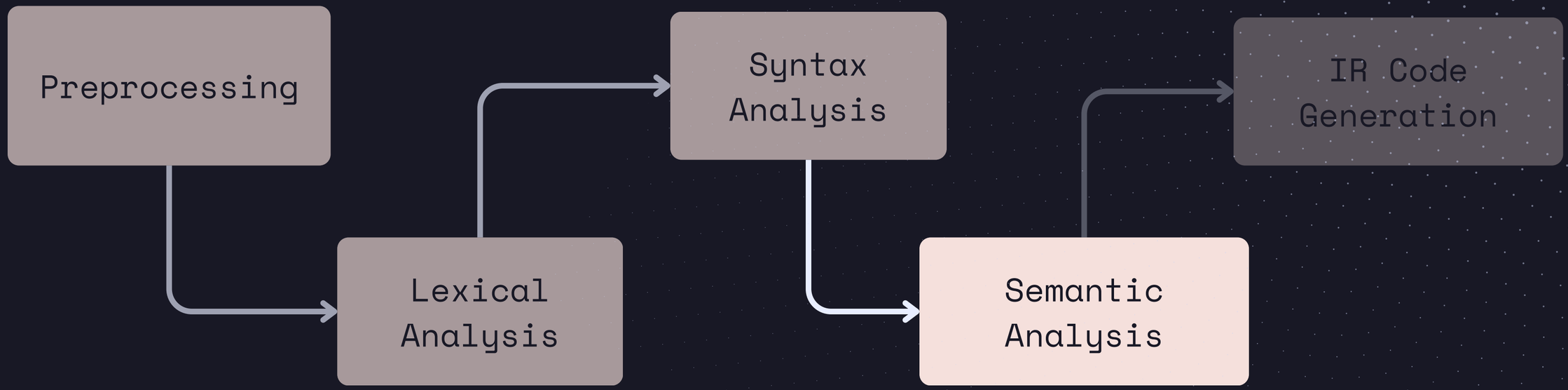

Preprocessing
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Code Generation



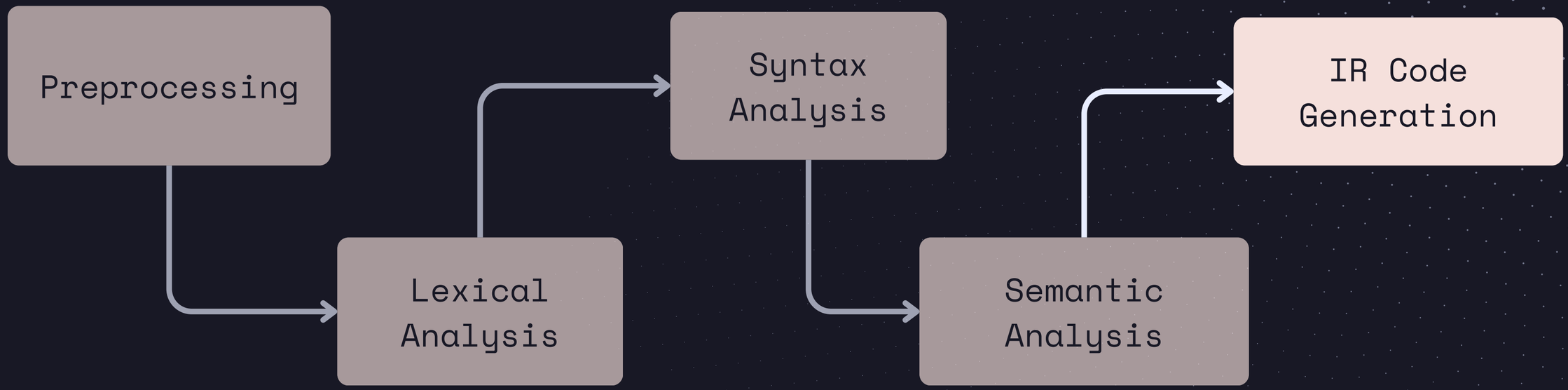
Syntax Tree

Source: Wikipedia

Front-end


Preprocessing
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Code Generation

Front-end


Preprocessing
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Code Generation

Intermediate Representation

Intermediate Representation

- A machine-independent representation of the source code generated by the frontend of the compiler.

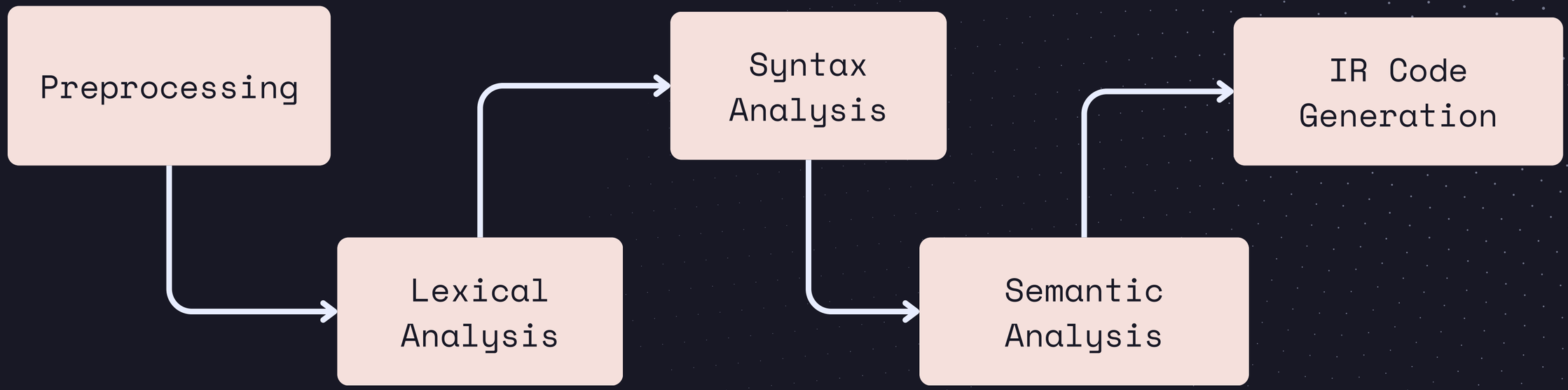
Intermediate Representation

- A machine-independent representation of the source code generated by the frontend of the compiler.
- Allows a single optimizer to be used across different machine architectures.

Intermediate Representation

- A machine-independent representation of the source code generated by the frontend of the compiler.
- Allows a single optimizer to be used across different machine architectures.
- Intermediate representation is written in an Intermediate Language, which differs from compiler to compiler and language to language. Examples include LLVM IR, C--, C, Rust IRs, etc.

Front-end


Preprocessing
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Code Generation

Middle-end

Middle-end

- Performs various optimisations on the IR code generated by the frontend.

Middle-end

- Performs various optimisations on the IR code generated by the frontend.

- Common optimisation techniques are:
  - Dead-code elimination
  - Constant propogation
  - Constant folding
  - Loop optimisation

Middle-end

- Performs various optimisations on the IR code generated by the frontend.

- Common optimisation techniques are:
  - Dead-code elimination
  - Constant propogation
  - Constant folding
  - Loop optimisation

- All optimisation performed in the middle-end are platform independent(as they are being done on the IR!)

Middle-end

Dead-code Elimination

Dead-code Elimination

- ‘Dead code’ refers to code that is never executed during runtime and has no impact on the program’s results.

Dead-code Elimination

- ‘Dead code’ refers to code that is never executed during runtime and has no impact on the program’s results.
- Dead code includes unreachable code like:

```
int some(int x, int y)
{
    int a = x + y;
    return a;
    a = 1;
    return 0;
}
```

Dead-code Elimination

- ‘Dead code’ refers to code that is never executed during runtime and has no impact on the program’s results.
- Dead code includes unreachable code like:

```
int some(int x, int y)
{
    int a = x + y;
    return a;
    a = 1;
    return 0;
}
```

- And redundant assignment operations like:

```
int sum(int x, int y)
{
    int a = x + y;
    a = 0;
    return a;
}
```

```c
int sum(int x, int y)
{
    int a = x + y;
    a = 0;
    return a;
}
```

Source code

```
- ; ModuleID = 'sum.ll'                                  - ; ModuleID = 'sum.ll'
  source_filename = "sum.c"                               source_filename = "sum.c"
  target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-  target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64
  target triple = "x86_64-pc-linux-gnu"                   target triple = "x86_64-pc-linux-gnu"

  ; Function Attrs: noinline nounwind uwtable            ; Function Attrs: noinline nounwind uwtable
  define dso_local i32 @sum(i32 noundef %0, i32 noundef %1) #0  define dso_local i32 @sum(i32 noundef %0, i32 noundef %1) #
    %3 = add nsw i32 %0, %1                               ----------------------------------------------------------
    ret i32 0                                               ret i32 0
  }                                                       }

  ; Function Attrs: noinline nounwind uwtable            ; Function Attrs: noinline nounwind uwtable
  define dso_local i32 @main() #0 {                       define dso_local i32 @main() #0 {
    %1 = call i32 @sum(i32 noundef 1, i32 noundef 2)        %1 = call i32 @sum(i32 noundef 1, i32 noundef 2)
+ +-- 14 lines: ret i32 0·····                           + +-- 14 lines: ret i32 0·····
```

IR with no DCE vs IR with DCE

Back-end

- Performs optimizations that depend on the CPU architecture.

- Performs code generation in three phases:
  - Instruction Selection
  - Instruction Scheduling
  - Register Allocation

Register Allocation

Register Allocation

- It is the process of assigning local automatic variables and expression results to a limited number of processor registers.

Register Allocation

- It is the process of assigning local automatic variables and expression results to a limited number of processor registers.

- It is a critical step in compilation and can significantly affect the efficiency of a program.

Register Allocation

- It is the process of assigning local automatic variables and expression results to a limited number of processor registers.

- It is a critical step in compilation and can significantly affect the efficiency of a program.

- A good register allocator can generate code orders
- of magnitude better than a bad register allocator.

Register Allocation

Register Allocation

```c
#include <stdio.h>

int sum(int x, int y)
{
    return x + y;
}

int main()
{
    sum(1, 2);
    return 0;
}

```

Source code

Register Allocation

```c
#include <stdio.h>

int sum(int x, int y)
{
    return x + y;
}

int main()
{
    sum(1, 2);
    return 0;
}

```

Source code

```
    .type   sum,@function
sum:                                    # @sum
    .cfi_startproc
# %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movl    %edi, -8(%rbp)
    movl    %esi, -4(%rbp)
    movl    -8(%rbp), %eax
    addl    -4(%rbp), %eax
    popq    %rbp
    .cfi_def_cfa %rsp, 8
    retq
```

The values of the x and y are passed through registers EDI and ESI

Register Allocation

```
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    movl    $0, -4(%rbp)
    movl    $1, %edi
    movl    $2, %esi
    callq   sum
    xorl    %eax, %eax
    addq    $16, %rsp
    popq    %rbp
    .cfi_def_cfa %rsp, 8
```

The values 1 and 2 are loaded into EDI and ESI registors

References

- Compiler - Wikipedia
- An Intro To Compilers
- Geeks for geeks
- CS31003 Compilers
- Optimizations in C++ Compilers

Thank You