GoCompiler

Your code may compile, but is it correct?
We'll tell you.

Team 05

Meneses Calderas Grecia Irais
Miramón Pérez Jocelyn
Membrilla Ramos Isaias Iñaki
Pérez Osorio Luis Eduardo
Salgado Cruz Emiliano Roman

RESULTS

STAGES FOR HELLOWORLD.GO

TAC / AST

```
Starting TAC Generation...
TAC Generation Success!
TAC saved to build folder
```

helloworld.log

helloworld.txt

ASM / C

```
Generating C code...
C code generated
ASM file generated

Assembling with NASM...
Successful Assembling:
```

helloworld.asm

helloworld.c

helloworld.exe

helloworld.obj

EXE

```
Linking with GCC...
Successful Linking:

--- EXECUTING helloworld.exe ---
Exit Code: 14
Output: Hello, World!

Compiling with GCC -> helloworld.exe...
Successful Compilation!

--- EXECUTING helloworld.exe ---

Program output:
Hello, World!
```

RESULTS

STAGES FOR HELLOWORLD.GO

TAC / AST

```
--------- THREE ADDRESS CODE (TAC): ----
FUNC main:
DATA str_0 = "Hello, World!"
CALL fmt.Println str_0
END_FUNC main
```

ASM

```
default rel

section .data
    fmt_int db "%d", 10, 0
    fmt_float db "%.2f", 10, 0
    fmt_string db "%s", 10, 0

    str_0: db "Hello, World!", 0

section .text
global main
extern printf, exit

main:
    push rbp
    mov rbp, rsp
    sub rsp, 32
    sub rsp, 32
    lea rcx, [fmt_string]
    lea rdx, [str_0]
    call printf
    add rsp, 32
    mov rsp, rbp
    pop rbp
    ret
```

C

```c
#include <stdio.h>
#include <stdbool.h>

int main() {
    printf("Hello, World!\n");
}
```

RESULTS

STAGES FOR OPERATION.GO

TAC / AST

```
Starting TAC Generation...
TAC Generation Success!
TAC saved to build folder
```

operation.log

operation.txt

ASM / C

```
Generating C code...
C code generated
ASM file generated

Assembling with NASM...
Successful Assembling:
```

operation.asm

operation.c

operation.exe

operation.obj

EXE

```
--- EXECUTING operation.exe -
Exit Code: 7
Output: Aritmeticas:
4
7
20
5
3
Relacionales:
0
1
0
1
0
1
IF simple:
a es 4
```

```
--- EXECUTING operation.exe ---

Program output:
Aritmeticas:
4
7
20
5
3

Relacionales:
0
1
0
1
0
1

IF simple:
a es 4
```

RESULTS

STAGES FOR OPERATION.GO

TAC / AST

```
--------- THREE ADDRESS CODE
FUNC main:
t0 = 2 ADD 2
a = t0
t1 = 10 SUB 3
b = t1
t2 = 4 MUL 5
c = t2
t3 = 20 DIV 4
d = t3
t4 = 15 % 4
e = t4
DATA str_0 = "Aritmeticas:"
CALL fmt.Println str_0
CALL fmt.Println a
CALL fmt.Println b
CALL fmt.Println c
CALL fmt.Println d
CALL fmt.Println e
```

```
x = 7
y = 10
DATA str_1 = "Relacionales:"
CALL fmt.Println str_1
t5 = x EQ y
CALL fmt.Println t5
t6 = x NE y
CALL fmt.Println t6
t7 = x GT y
CALL fmt.Println t7
t8 = x LT y
CALL fmt.Println t8
t9 = x GE y
CALL fmt.Println t9
t10 = x LE y
CALL fmt.Println t10
DATA str_2 = "IF simple:"
CALL fmt.Println str_2
t11 = a EQ 4
IF_FALSE t11 GOTO L0
DATA str_3 = "a es 4"
CALL fmt.Println str_3
GOTO L1
LABEL L0:
DATA str_4 = "a NO es 4"
CALL fmt.Println str_4
LABEL L1:
END_FUNC main
```

ASM

```
section .data
    fmt_int db "%d", 10, 0
    fmt_float db "%.2f", 10, 0
    fmt_string db "%s", 10, 0

    str_0: db "Aritmeticas:", 0
    str_1: db "\nRelacionales:", 0
    str_2: db "\nIF simple:", 0
    str_3: db "a es 4", 0
    str_4: db "a NO es 4", 0

section .bss
    t0 resd 1
    a resd 1
    t1 resd 1
    b resd 1
```

```
main:
    push rbp
    mov rbp, rsp
    sub rsp, 160
    sub rsp, 32
    ; t0 = 2 + 2
    mov dword [t0], 4
    ; a = t0
    mov eax, [t0]
    mov [a], eax
    ; t1 = 10 - 3
    mov dword [t1], 7
    ; b = t1
    mov eax, [t1]
    mov [b], eax
    ; t2 = 4 * 5
    mov dword [t2], 20
```

```
L0:
    ; CALL fmt.Println str_4
    ; PRINTLN: str_4
    sub rsp, 32
    lea rcx, [fmt_string]
    lea rdx, [str_4]
    call printf
    add rsp, 32
    ; Fin de println
L1:
    ; END_FUNC
    mov rsp, rbp
    pop rbp
    ret
```

C

```
#include <stdio.h>
#include <stdbool.h>

int main() {
    int a = (2 + 2);
    int b = (10 - 3);
    int c = (4 * 5);
    int d = (20 / 4);
    int e = (15 % 4);
    printf("Aritmeticas:\n");
    printf("%d\n", a);
    printf("%d\n", b);
    printf("%d\n", c);
    printf("%d\n", d);
    printf("%d\n", e);
```

```
    int x = 7;
    int y = 10;
    printf("\nRelacionales:\n");
    printf("%d\n", (x == y));
    printf("%d\n", (x != y));
    printf("%d\n", (x > y));
    printf("%d\n", (x < y));
    printf("%d\n", (x >= y));
    printf("%d\n", (x <= y));
    printf("\nIF simple:\n");
    if ((a == 4)) {
        printf("a es 4\n");
    } else {
        printf("a NO es 4\n");
    }
}
```

ADVANTAGES AND DIFFERENTIATORS

Complete Pipeline

The program not just checks lexical, syntax, and semantic correctness, now it produces excecutable code.

Modular Design

Each phase of the flow is fully separated into modules, facilitating maintenance and extensibility

End-to-end compiler

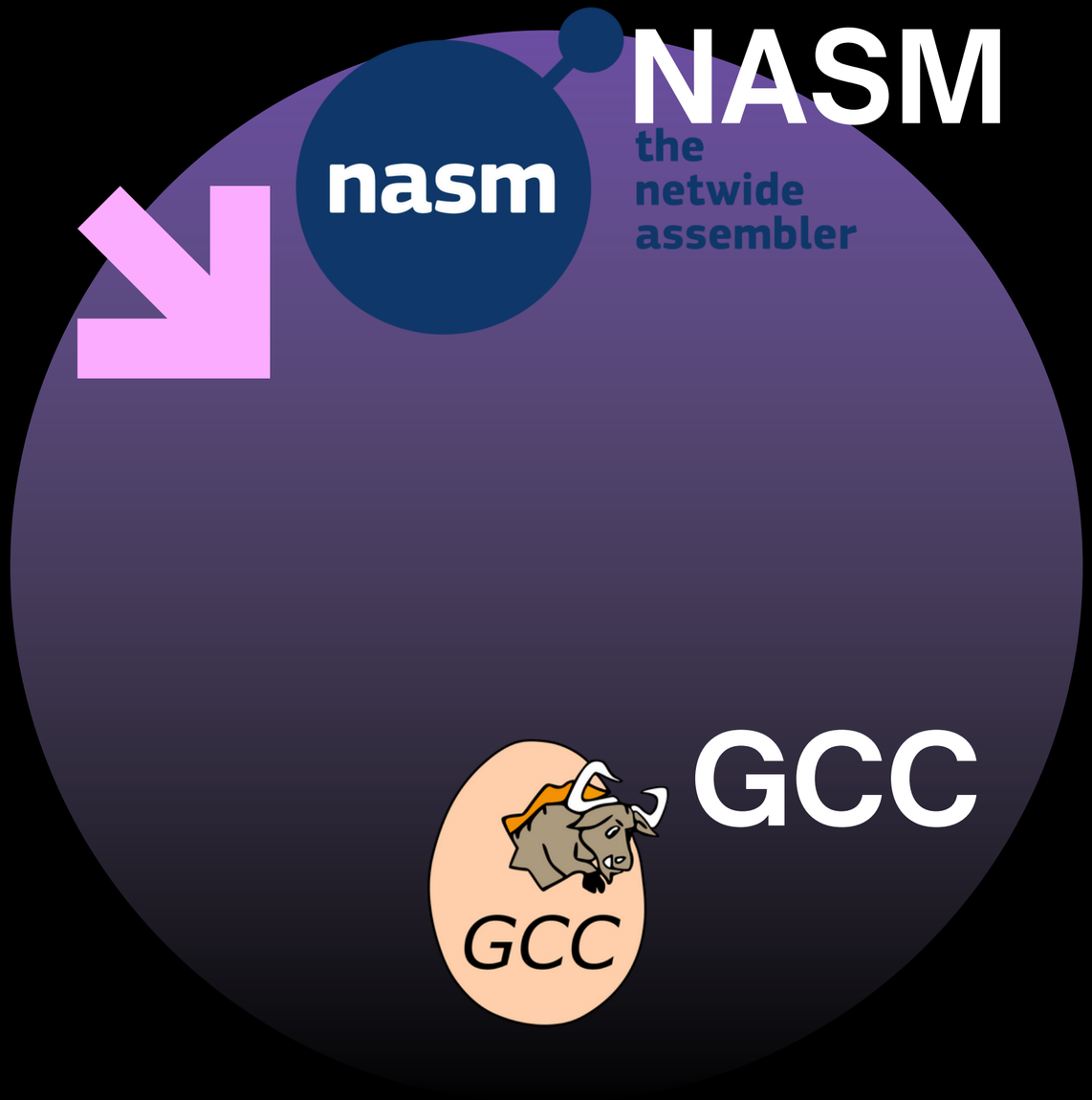
The project goes from Go source code with AST-based path and TAC-based path.

Files for every stage

Each compilation run stores the results of all stages, so is simple to inspect how a source code evolves.

Optimization

Using Three-Adress Code makes it easier to add compiler optimizations and future implementations.



The assembly pipeline utilizes standard NASM syntax for its intermediate representation, this assembler is also used to generate object code.

For both pipelines GCC serves as the llinker for our object code, for the C pipeline it alse serves as a secondary compiler and object code generator.

TAC

The assembly pipeline utilizes standard NASM syntax for its intermediate representation, this assembler is also used to generate object code.

```
def new_temp(self):
    temp = f"t{self.temp_counter}"
    self.temp_counter += 1
    return temp

def new_label(self):
    label = f"L{self.label_counter}"
    self.label_counter += 1
    return label
```

TAC

No.	TAC Form	Usage in the Provided Code
1	`x = constant`	Direct constant assignment.
2	`GOTO L`	Unconditional jump to label L.
3	`LABEL L:`	Label definition for control flow targets.
4	`x = y`	Simple assignment between variables.
5	`t = a op b`	Binary operation with assignment to a temporary.
6	`x = t`	Assignment of temporary result to a variable.
7	`x = x op constant`	In-place operation with a constant.
8	`IF_TRUE t GOTO L`	Conditional jump if the temporary is true.
9	`IF_FALSE t GOTO L`	Conditional jump if the temporary is false.
10	`RETURN x`	Return value from a function.
11	`END_FUNC name`	End of function definition.
12	`FUNC name:`	Start of function definition.
13	`CALL func arg`	Function call without storing a result.
14	`DATA name`	For temporal strings assigments

TAC

```
// IF simple
fmt.Println("\nIF simple:")
if (a == 4) {
    fmt.Println("a es exactamente 4")
} else {
    fmt.Println("a NO es 4")
}
```

```
DATA str_2 = "\nIF simple:"
CALL fmt.Println str_2
t11 = a EQ 4
IF_FALSE t11 GOTO L0
DATA str_3 = "a es exactamente 4"
CALL fmt.Println str_3
GOTO L1
LABEL L0:
DATA str_4 = "a NO es 4"
CALL fmt.Println str_4
LABEL L1:
END_FUNC main
```

TAC

```go
func test() {
    var n, n1 int = 5, 1
    for i := 2; i <= n; i++ {
        n1 = n1 + i
    }
}
```

```
THREE ADDRESS CODE (TAC):
==================================================
FUNC test:
n = 5
n1 = 1
i = 2
GOTO L2
LABEL L0:
t0 = n1 ADD i
n1 = t0
i = i + 1
LABEL L2:
t1 = i LE n
IF_TRUE t1 GOTO L0
LABEL L1:
END_FUNC test
```

UNDER THE HOOD
THE LOGICAL ENGINE

How our compiler “understads what the code actually means

VISITOR PATTERN

“The Inspector”

- Walks the tree
- Recognizes meaning
- Enables growth
- Generates the C and TAC IRs

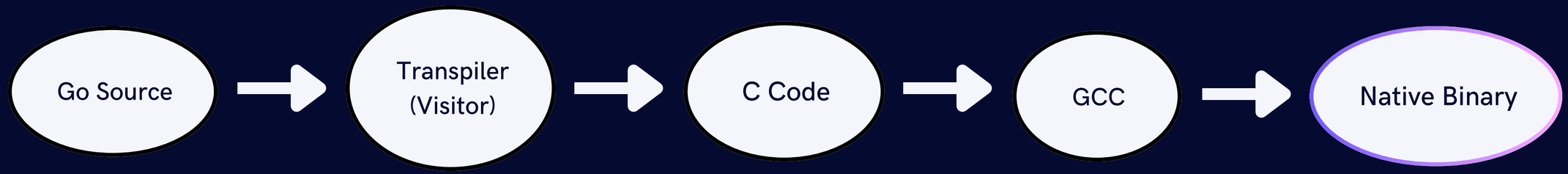
SYMBOL TABLE

“The Memory”

- Keeps declarations
- Tracks types and scope
- Prevents misuse
- Used for assembly IR

TARGET CODE GENERATION

NATIVE TRANSPILATION



- Source-to-Source Compilation (Transpiler)
- Leverages GCC for industrial-grade optimization
- Implements the Visitor Pattern for efficient AST traversal

TARGET CODE GENERATION

ASM CODE GENERATION

Two-pass compiler implementation

Leverages NASM to generate object code and GCC to link it

Generates assembly code from the TAC IR and the data from the symbol table

BRIDGING THE SEMANTIC GAP

SCOPE ISOLATION

```go
package main

import "fmt"

func fibonacciIterative(n int) int {
    if n <= 1 {
        return n
    }
    var n2, n1 int = 0, 1
    for i := 2; i <= n; i++ {
        temp := n1
        n1 = n1 + n2
        n2 = temp
    }
    return n1
}

func main() {
    fmt.Println(fibonacciIterative(9)) // Output: 34
}
```

fib.go

```c
#include <stdio.h>
#include <stdbool.h>

int fibonacciIterative(int n) {
    if ((n <= 1)) {
        return n;
    }
    int n2 = 0;
    int n1 = 1;
    {
        int i = 2;
        while ((i <= n)) {
            int temp = n1;
            n1 = (n1 + n2);
            n2 = temp;
            i++;
        }
    }
    return n1;
}
int main() {
    printf("%d\n", fibonacciIterative(9));
}
```

fib.c

Smart Memory Management for Control Flow

Auto-detects types for short declarations (:=)

Polymorphic mapping of fmt.Println to printf

Thank **You**