Ingeniería en Computación

Lenguajes de Programación

Mini-Go Compiler

Profesor:

Oscar Mario Víquez Acuña

Grupo 50, 51

Estudiantes:

Kevin Vinicio Varela Rojas

Anthony Andrés Jiménez Zamora

Fabián Rojas Ugalde

Sede San Carlos

03/06/2024

La fase final se centra en la implementación de la librería LLVM, para que permita al compilador tener funcionamiento. A continuación, se presenta una descripción exhaustiva de la solución e implementación de la carpeta "encoder", la cual contiene los archivos: `BlockVariableTable.go`, `EncoderLLVM.go`, `generalStack.go`. Estos archivos son esenciales para la generación del código intermedio y su traducción a código máquina optimizado.

Solución e Implementación

BlockVariableTable

El archivo `BlockVariableTable.go` gestiona la tabla de variables por bloques, una estructura fundamental para el manejo del ámbito (scope) de las variables dentro de los bloques de código. La implementación de esta tabla permite mantener un registro preciso de las variables declaradas y utilizadas en cada bloque, facilitando así la asignación de memoria y la optimización del acceso a las variables durante la compilación.

La estructura de datos principal en este archivo es un mapa que relaciona el nombre de las variables con sus respectivas representaciones en LLVM. Las funciones incluidas permiten agregar nuevas variables, buscar variables existentes y eliminar variables al salir de un bloque. Esto asegura que cada bloque tenga acceso sólo a las variables definidas en su ámbito, manteniendo así la integridad del código y evitando colisiones de nombres.

EncoderLLVM

`EncoderLLVM.go` es el corazón del proceso de compilación. Este archivo contiene las funciones encargadas de convertir el código de alto nivel a instrucciones de LLVM. Utilizando la API de LLVM, se generan las representaciones intermedias necesarias para la compilación y optimización del código.

La implementación incluye la creación de los módulos LLVM, la definición de funciones y bloques básicos, y la emisión de instrucciones específicas como operaciones aritméticas o asignaciones. Este archivo también se encarga de la generación del archivo ejecutable final.

GeneralStack

El archivo `generalStack.go` proporciona una pila generalizada utilizada para manejar diversas estructuras de datos durante la compilación. Esta pila maneja diferentes contextos, tales como bucles `for`, funciones y estructuras condicionales como `if`.

Manejo de Bucles

Dentro de `generalStack.go`, se implementa el manejo de la pila de bucles. Esta estructura de datos es crucial para gestionar las múltiples capas de bucles anidados. La implementación incluye funciones para apilar y desapilar información relevante a cada bucle, como etiquetas de inicio y fin, y contadores de iteraciones.

Manejo de Funciones

El manejo de la pila de funciones es otro aspecto clave cubierto en `generalStack.go`. Cada entrada en la pila contiene información sobre la función actual, incluidos sus parámetros, variables locales y puntos de retorno

Manejo de Condicionales

Finalmente, `generalStack.go` también gestiona las estructuras condicionales `if`. Similar a las pilas de bucles, esta estructura permite al compilador manejar correctamente los bloques de código condicional, manteniendo información sobre las etiquetas de inicio y fin de cada bloque `if`, `else if` y `else`.

Resultados obtenidos:

Aspecto	Realizado	No Realizado
Declaraciones y usos de variables de tipos simples y complejas	Se puede declarar los tipos simples como int, string, bool, rune	No se pueden reasignar los valores a las variables dentro de funciones
Declaraciones y usos de arreglos de enteros	Se pueden declarar arreglos	No se puede asignar valores a los índices
Declaraciones y usos de métodos	Se pueden declarar funciones	No se pueden llamar funciones dentro de funciones
Instrucciones de control de flujo	Se pueden declarar ifs y loops	No se pueden declarar for dentro de ifs
Println, len	Se puede imprimir ints y utilizar la función len	No se pueden imprimir strings, y los floats se imprimen con numeros más grandes
Comparar expresiones	Si se pueden comparar expresiones	
Documentación	La documentación se realizó al completo	
Manual de pruebas	Se realizó un video con las pruebas del correcto funcionamiento del compilador	

Manual de pruebas:

Código miniGo

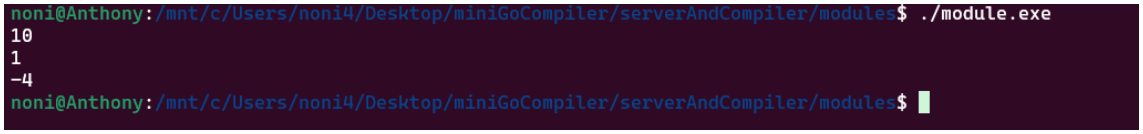
```
package main;

var global = 20;

var (palabra string; puntoFlotante = 10.2;

var a, b, c = 34,"Hola",30.5;

func suma(i int) int{

    i = 25;

    var sumar int = i + 10;

    return sumar;
};

func imprimir(){
    println(55);
};
```

```
func main() int{

    var comparison = 1!=2;

    var array [10]int;

    var lenVariable int = len(array);

    println(lenVariable);

    var cond = 10;

/*
    for i = 0; i < cond ;i++{
        println(1111);
    };
*/

    if (cond <= global){
        println(1);
    } else{
        println(2);
    };

    if (cond >= 50){
        println(23);
    } else if (cond == 10){
        var compExp = ((5*2)+2)-(8/2)*4;
        println(compExp);
    };

    return 0;
};
```

Código .ll

```
@0 = global [4 x i8] c"%d\0A\00"
@1 = global [4 x i8] c"%f\0A\00"
@2 = global [4 x i8] c"%s\0A\00"
@3 = global [4 x i8] c"%c\0A\00"
@global = global i32 20
@palabra = global i8* zeroinitializer
@puntoFlotante = global float 0x4024666660000000
@a = global i32 34
@b = global [6 x i8] c"\22Hola\22"
@c = global float 30.5

declare i32 @printf(i8* %0, ...)

define i32 @suma(i32 %0) {
1:
    %2 = alloca i32
    store i32 %0, i32* %2
    store i32 25, i32* %2
    %3 = load i32, i32* %2
    %4 = add i32 %3, 10
    %5 = alloca i32
    store i32 %4, i32* %5
    %6 = load i32, i32* %2
    %7 = add i32 %6, 10
    %8 = load i32, i32* %5
    ret i32 %8
}

define void @imprimir() {
0:
    %1 = call i32 (i8*, ...) @printf([4 x i8]* @0, i32 55)
    ret void
}

define i32 @main() {
0:
    %1 = icmp ne i32 1, 2
    %2 = alloca i1
    store i1 %1, i1* %2
    %3 = load i1, i1* %2
    %4 = alloca [10 x i32]
    store [10 x i32] zeroinitializer, [10 x i32]* %4
    %5 = alloca i32
    store i32 10, i32* %5
    %6 = load i32, i32* %5
    %7 = call i32 (i8*, ...) @printf([4 x i8]* @0, i32 %6)
    %8 = alloca i32
    store i32 10, i32* %8
    %9 = load i32, i32* %8
    %10 = load i32, i32* %8
    %11 = icmp sle i32 %10, 20
    br i1 %11, label %12, label %14

12:
    %13 = call i32 (i8*, ...) @printf([4 x i8]* @0, i32 1)
    br label %16

14:
    %15 = call i32 (i8*, ...) @printf([4 x i8]* @0, i32 2)
    br label %16

16:
    %17 = load i32, i32* %8
    %18 = icmp sge i32 %17, 50
    br i1 %18, label %19, label %21

19:
    %20 = call i32 (i8*, ...) @printf([4 x i8]* @0, i32 23)
    br label %34

21:
    %22 = load i32, i32* %8
    %23 = icmp eq i32 %22, 10
    br i1 %23, label %24, label %34

24:
    %25 = mul i32 5, 2
    %26 = add i32 %25, 2
    %27 = sdiv i32 8, 2
    %28 = mul i32 %27, 4
    %29 = sub i32 %26, %28
    %30 = alloca i32
    store i32 %29, i32* %30
    %31 = load i32, i32* %30
    %32 = load i32, i32* %30
    %33 = call i32 (i8*, ...) @printf([4 x i8]* @0, i32 %32)
    br label %34

34:
    ret i32 0
}
```

Resultado: (Compilado en Linux Ubuntu)



Video explicativo: https://youtu.be/NlSxTWceM2A

Conclusión

Se logró realizar la mayoría de lo pedido, el compilador cumple con su funcionamiento, aunque presenta fallas que antes se mencionaron. Es importante destacar que, a pesar de los avances y la implementación exitosa de la mayoría de los componentes, aún existen áreas que requieren atención y mejora. La base establecida en este proyecto proporciona una sólida plataforma sobre la cual se pueden realizar optimizaciones y correcciones adicionales.

En conclusión, este proyecto demuestra la viabilidad de crear un compilador funcional con Go y LLVM, destacando tanto los logros como los desafíos encontrados durante el desarrollo.

Bibliografía:

ChatGPT. (n.d.). *ChatGPT*. Ayuda en minigo. Retrieved junio, 2024, from https://chat.openai.com/

LLVM Project. (n.d.). *LLVM: A collection of modular and reusable compiler and toolchain technologies*. Retrieved June 3, 2024, from https://llvm.org/

ANTLR. (n.d.). *ANTLR: ANother Tool for Language Recognition*. Retrieved June 3, 2024, from https://www.antlr.org/