Taller de Diseño de Software - Reporte

1. Estructura y consideraciones del Proyecto.

- Estructuralmente, dividimos el proyecto en tres grandes módulos *frontend*, *backend* y *error-handling*. En un principio hacíamos todo dentro de src/, pero luego vimos que modularizando podíamos trabajar de manera más organizada y precisa.
- Utilizamos git & github como sistema de versionado y haciendo principalmente uso de ramas para distintas features que implementamos.
- Implementamos Integración Continua (CI) para que en cada push se ejecute la suite de test previniendo así commits que puedan ocasionar daños colaterales a otras funcionalidades.
- Para legibilidad de código, optamos por evitar el uso de punteros a void e incorporamos ciertos estándares en nuestros archivos (estilo de Google provisto por clang).
- Sabíamos que en algún punto íbamos a enfrentar ciertos desafíos que iban requerir debuggear, encontrar o identificar memory leaks,entre otros, es por ello que optamos por configurar un entorno único de desarrollo donde incorporamos:
    - **Clang as plugin**: (formatea automáticamente todos los archivos .c siguiendo el estándar elegido)
    - **GDB**: como debugger utilizado tanto en líneas de comando como también mediante una interfaz.
    - **Dot**: para visualizar los AST de manera más clara.
    - **CMake**: como herramienta para buildear el proyecto y resolver '*path dependencies*' entre los distintos archivos de cabecera, flex y bison.
    - **Valgrind**: como herramienta para detectar los problemas de memoria (potenciales memory leaks).
    - Entre otros.

2. Análisis léxico - sintáctico y semántico.

2.1 Relevante.

- Permitimos variables globales, las mismas solo pueden ser inicializadas como constantes.
- Recursión y uso de funciones externas, es posible ya que cada parámetro y cuerpo de la función tienen sus propios niveles y se guarda el identificador previo al procesamiento del bloque.

- El chequeo *"semántico"* se encuentra dividido, realizando la mayor parte del mismo recorriendo el árbol después de ser construido (En el **parser.y** solo se chequea, por ejemplo, que cuando se llama a una función con parámetros tenga la misma cantidad que en su declaración y que la misma exista, lo mismo para otros identificadores).
- Variables globales al no permitir ser inicializadas con expresiones, esto simplifica que la inicialización no debe ser dinámica (obvio tiene su limitante, que se describe más abajo).
- Se chequea que las funciones tengan al menos un return. Como también que el mismo debe ser del mismo tipo de la función .

2.2 Limitaciones.

- Todas las declaraciones deben ser inicializadas, esto limita el uso de variables globales, sería más útil poder inicializarlas sin valor y luego asignarlos cuando sea necesario, eso evitará chequeos innecesarios.
- Las variables globales sólo pueden ser inicializadas por literales.
- Actualmente solo soportamos dos tipos de datos básicos (int y bool), sería interesante la incorporación de flotantes y carácteres, ya que enriquecen mucho la legibilidad de los programas además de poder abordar problemas matemáticos con más precisión y poder resolver problemas que involucren strings.
- No se chequea la alcanzabilidad del return en una función, por lo que pueden existir caminos donde no se alcance un return, en esta caso la función retorna el valor que se encuentre en rax.

3. Generación de código 3 direcciones(TAC) y assembly GNU X86

3.1 Relevante:

- Hacemos uso de macros para poder obtener rápidamente los diferentes códigos de operaciones.
- Se diferencia la asignación de valores constantes, variables locales y globales lo que simplifica la generación del assembly.
- Se trata las operaciones de mult y and como equivalentes a la hora de generar el assembler al igual que el or y add.
- Por cómo encaramos la generación de assembly, la recursión, el while, y muchos operadores *"similares"* (< con >, = , * con &&, etc..) salieron casi o totalmente "*gratis*" en el sentido que no fue necesario programar mucho más.
- Implementación de un manejo de offset único/independiente para cada función. Esta característica permite a cada función gestionar su propio espacio de memoria de manera aislada, garantizando que las variables locales y los parámetros de cada función se manejen de manera segura y eficiente.

3.2 Limitaciones

- Las funciones solo pueden tener un máximo de 6 parámetros que se corresponden con los registros : rdi, rsi, rdx, rcx, r8, r9 .
- Para simplificación del proyecto actualmente cada offset necesario se guarda en un *“attributes”* que se encuentra en cada nodo, algunos nodos no necesitan ciertos campos, para mayor modularización se podría pensar hacer un struct de cada nodo particular.
- No se optimiza el assembler generado ni se utiliza un director de registros, estas son *features* interesantes para plantear en una futura versión.

4. Manejo de errores.

En el diseño de nuestro compilador, hemos puesto especial énfasis en el manejo de errores, un aspecto clave que atraviesa todas las fases del desarrollo. Nuestro objetivo es informar los errores de la manera más precisa y clara posible. Para lograr esto, hemos adoptado un enfoque sistemático:

4.1 Definición de Constantes de Error

Para evitar el *"hardcodeo"* de mensajes de error, definimos constantes en **error.h**. Estas constantes están diseñadas para ser descriptivas y autoexplicativas. Algunos ejemplos incluyen:

#define TYPE_ERROR_OPERATION "Type Error in operation %s : left-hand side has type %s but right-hand side has type %s"

#define TYPE_ERROR_ASSIGNMENT "Type Error in : variable of type %s cannot be assigned a value of type %s"

#define TYPE_ERROR_DECLARATION "Type Error in : variable declared as %s but initialized with a value of type %s"

Estas constantes están preparadas para la interpolación, lo que significa que cuando se detecta un error, podemos insertar dinámicamente los operandos o identificadores específicos involucrados para proporcionar un informe detallado y útil.

4.2 Mejora en el Reporte de Errores de Identificadores

En casos donde se encuentran errores relacionados con variables no declaradas o no encontradas, hemos implementado un algoritmo que realiza un matcheo de nombres para sugerir(inferir) posibles identificadores correctos. Por ejemplo, considera el siguiente fragmento de código:

```
Program{

   int gauss_sum(int n){
```

```
        int sum = n*(n+1)/2;
        return sum;
    }
    int main(){
        int n = 100;
        int sum = gauss_sum(n);
        return suma;
    }
}
```

El compilador informará:

Identifier 'suma' not declared;

Did you mean: 'sum'? at line 11