72.39

Autómatas, Teoría de Lenguajes y Compiladores

2º Cuatrimestre del 2023

Parte II: Backend

Integrantes:

- **Ezequiel Pérez** **61475**
- **Valentín Ye Li** **61011**
- **Gonzalo Baliarda** **61490**

Tabla de contenidos

Autómatas, Teoría de Lenguajes y Compiladores..... 0
Tabla de contenidos..... 1
Introducción y objetivos..... 2
Descripción del desarrollo..... 2
Arquitectura del compilador..... 2
Opciones de línea de comandos..... 3
Lexer..... 3
Parser..... 4
Tabla de Símbolos..... 5
Type Checking..... 6
Generación de código..... 6
Compilación y ejecución del programa de salida..... 7
Remix..... 7
Foundry..... 8
Dificultades encontradas..... 8
Futuras extensiones..... 9
Referencias..... 10

Introducción y objetivos

El objetivo de este trabajo es desarrollar un lenguaje que permita crear Smart Contracts que corran sobre la Ethereum Virtual Machine (EVM). El lenguaje será similar a Solidity, el lenguaje más popular para crear Smart Contracts en la actualidad, heredando en parte su sintaxis y añadiendo a la vez nuevas funcionalidades.

La idea es que este nuevo lenguaje permita hacer más fáciles y seguras algunos tipos de operaciones comunes.

Descripción del desarrollo

Arquitectura del compilador

El siguiente diagrama muestra la arquitectura general del compilador.

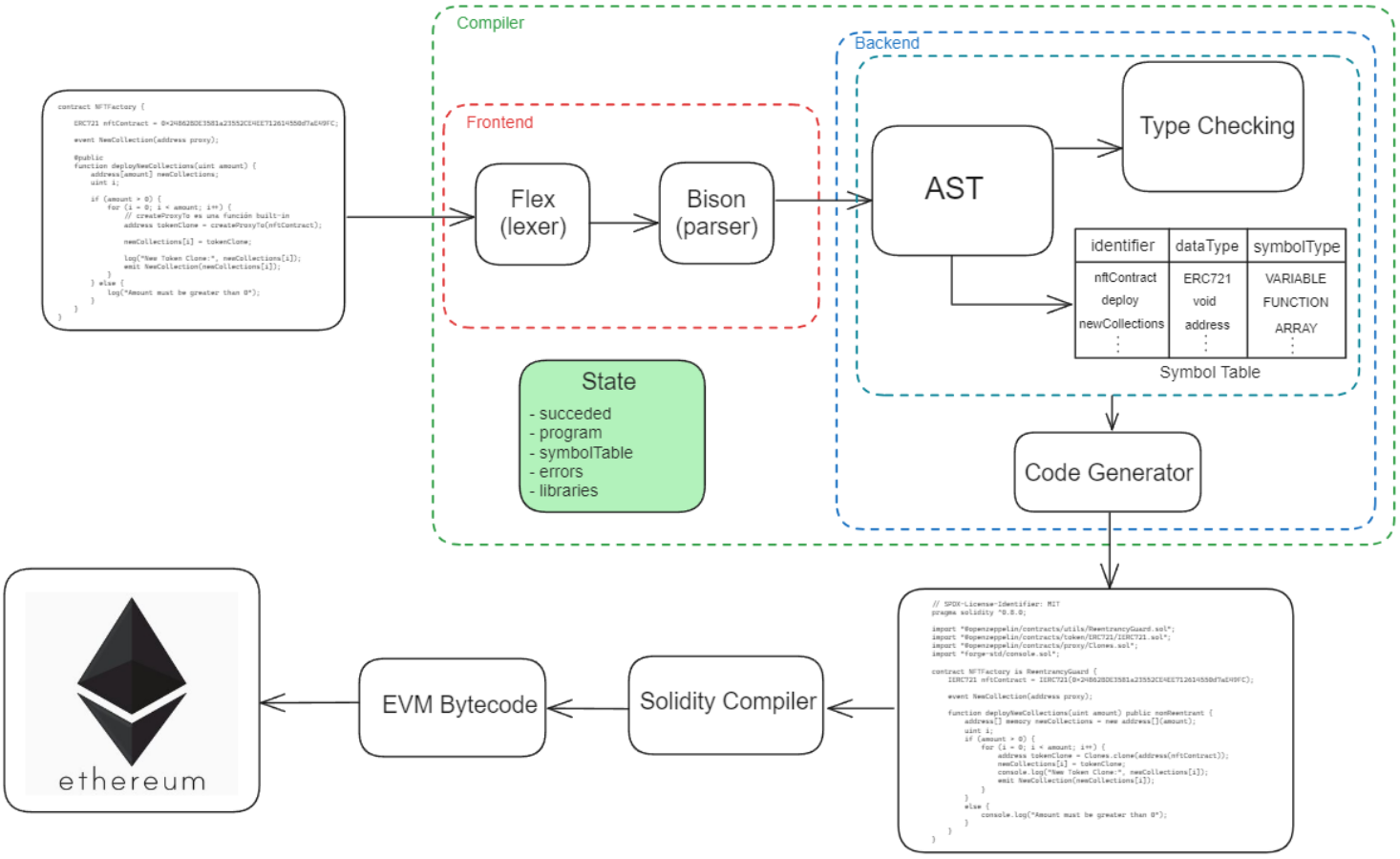

Figura 1: arquitectura del compilador.

Como parte del estado del compilador, se tienen los siguientes campos:

- **Succeeded**: indica si la compilación fue exitosa o no.
- **Program**: puntero el nodo raíz del AST.

- **SymbolTable**: puntero a la tabla de símbolos.
- **Errors**: arreglo que contiene todos los errores que surgieron al compilar cierto programa. Cada error, indicará los detalles del mismo y el número de línea en el que se produjo.
- **Libraries**: arreglo que se va actualizando durante el armado del AST, indicando qué librerías externas deben incluirse durante la fase de generación de código.

Opciones de línea de comandos

El compilador permite recibir opciones por línea de comandos para definir la configuración a usar. Las opciones disponibles son:

- ***-o <output_file>*:** especifica el nombre del archivo de salida con el programa compilado. El default es *"out.sol"*.
- ***-i <indent_size>***: especifica el tamaño de la indentación en el programa de salida. El default es 4.
- ***-t***: utilizar "TABS" para la indentación en lugar de espacios.
- ***-m***: no indentar el programa de salida.
- ***-h***: imprime información sobre las opciones de línea de comandos.

Lexer

Implementado en Flex, tiene soporte para reconocer diversos patrones relevantes como ser: números en notación científica, claves públicas de Ethereum, comentarios de línea y de bloque, múltiples tipos de datos, decoradores, y operaciones aritméticas y lógicas.

Hay cinco métodos que son los principales para manejar el armado de cada *token* identificado:

```
token TokenPatternAction(const char *lexeme, token token) {
    yylval.token = token;
    return token;
}
```

Figura 2: acción que maneja el armado de tokens que no necesitan guardar una representación numérica o de texto, sino solo su tipo.

```
token StringValuePatternAction(const char *lexeme, const int length, token token) {
    yylval.string = strndup(lexeme, length);
    return token;
}
```

Figura 3: acción que maneja el armado de tokens que guardan una representación textual, tal cual son leídos.

```
token DecoratorPatternAction(const char *lexeme, const int length) {
    yylval.string = strndup(lexeme + 1, length - 1); // ignore leading '@'
    return DECORATOR;
}
```

Figura 4: acción que maneja el armado de tokens de tipo *decorator* (eg. @public), guardando sólo el decorator en sí e ignorando el @ inicial.

```
token BooleanPatternAction(const char *lexeme, const int length) {
    char *lexemeCopy = strndup(lexeme, length);
    yylval.integer = strncmp(lexemeCopy, "true", length) == 0;
    free(lexemeCopy);
    return BOOLEAN;
}
```

Figura 5: acción que maneja el armado de tokens de tipo *boolean*, guardando el valor como un entero binario 0 (falso) o 1 (verdadero).

```
token IntegerPatternAction(const char *lexeme, const int length) {
    char *lexemeCopy = strndup(lexeme, length);
    yylval.integer = atoi(lexemeCopy);
    free(lexemeCopy);
    return INTEGER;
}
```

Figura 6: acción que maneja el armado de tokens de tipo numérico, guardando su valor como metadata.

Parser

Implementado en Bison, define la gramática del lenguaje desarrollado. Cada vez que se aplica una reducción, se crea un nodo del AST, y estos se van encadenando hasta llegar al nodo *Program* que representa la raíz del árbol de sintaxis.

En este punto, durante el armado del AST, se hacen también los chequeos correspondientes al análisis semántico, que involucran la tabla de símbolos y el chequeo de tipos.

Por ejemplo, la siguiente acción es la encargada de generar un nodo del AST que representa un llamado a función:

```
FunctionCall *FunctionCallGrammarAction(char *identifier, Arguments *arguments) {
    if (!symbolExists(identifier) && !isBuiltInFunction(identifier))
        addError(sprintf(ERR_MSG, "Function `%s` does not exist", identifier));

    FunctionCall *functionCall = calloc(1, sizeof(FunctionCall));

    if (isBuiltInFunction(identifier))
        functionCall->type = getBuiltInType(identifier);
    else
        functionCall->type = arguments->type == ARGUMENTS_EMPTY ? FUNCTION_CALL_NO_ARGS :
        FUNCTION_CALL_WITH_ARGS;

    functionCall->identifier = identifier;
    functionCall->arguments = arguments;

    if (typeFunctionCall(functionCall) == -1)
        addError(sprintf(ERR_MSG, "%s is not callable", identifier));

    return functionCall;
}
```

Figura 7: acción que maneja el armado de un nodo del AST, correspondiente a un llamado a función.

Se puede apreciar que la acción verifica que el nombre de la función corresponda a un identificador guardado en la tabla de símbolos y del tipo “función”, o bien a un identificador correspondiente a una función built-in del lenguaje.

Tabla de Símbolos

Estructura de datos utilizada para almacenar identificadores únicos. Estos identificadores pueden ser variables, funciones, eventos o arreglos.
La tabla está compuesta por 3 campos:

- **Identifier**: corresponde al nombre de la variable, función, evento o arreglo. Este campo es único y no se puede repetir en la tabla.
- **DataType**: tipo de dato básico del identificador. Por ejemplo, si se tiene una variable de tipo entero, este tipo será *uint*. Si se tiene un arreglo de enteros, de igual forma será *uint*.
- **SymbolType**: tipo de símbolo del identificador. Puede ser *variable*, *function*, *event* o *array*.

Internamente, esta estructura fue implementada como una tabla de hash, haciendo uso de la librería uthash:

```c
typedef struct SymbolTableEntry {
    char *identifier;  // hash key
    DataTypeType type;
    SymbolType symbolType;
    UT_hash_handle hh; // makes this structure hashable
} SymbolTableEntry;
```

Figura 8: entrada de la tabla de símbolos.

Type Checking

En este módulo se implementaron chequeos de tipos para diferentes operaciones del lenguaje. Para cada tipo de nodo del AST, se implementó una función que retorna su tipo, o un tipo especial que indica que el mismo es inválido.

Este proceso se llevó a cabo durante la creación de cada nodo del AST, con el fin de poder indicar con precisión el número de línea en donde se produjo el error, el cual es informado por Bison al procesar una reducción.

En el caso de construcciones válidas entre tipos diferentes, se aplica coerción de tipos. Para definir qué tipos son coaccionables entre sí, utilizamos las siguientes cadenas de coacciones:

uint < int
ERC20 < address
ERC721 < address

Por ejemplo, las comparaciones entre *uint* e *int* se toman como válidas, así como también la asignación de elementos de tipo *address* a variables del tipo *ERC20* o *ERC721* (que son estándares de contratos de la red de Ethereum).

Generación de código

La interfaz de este módulo consta de solamente un método: *Generator*.

```c
void Generator() {
    includeDependencies(state.program);
    generateProgram(state.program);
}
```

Figura 9: punto de entrada para la generación de código.

En el primer paso, se incluye el código de *Runtime*, es decir, que no corresponde a la traducción del AST directamente. En este punto, se escriben en el archivo de salida la licencia, la versión del compilador y los *imports* necesarios para la ejecución del programa final.

Luego, se procede a la generación del programa en sí. Este es un proceso recursivo, arrancando desde la raíz del árbol y expandiéndose luego a sus nodos hijos, hasta llegar a los nodos hoja (correspondientes a los símbolos terminales de la gramática).

```
static void generateProgram(Program *program) {
    output("contract %s is ReentrancyGuard ", program->contract->identifier);
    output("{");
    generateContractInstructions(program->contract->block->instructions);
    output("}\n");
}
```

Figura 10: generación de código del nodo raíz del AST.

Compilación y ejecución del programa de salida

La salida del compilador será un programa con sintaxis de Solidity. Para compilarlo y ejecutarlo, se recomiendan las siguientes opciones:

Remix

Remix es un IDE para crear, compilar y deployar smart contracts. Para compilar y deployar un contrato generado por nuestro compilador los pasos a seguir son:

1. Abrir Remix.
2. Importar el archivo de salida del compilador.
3. Compilar el archivo utilizando la versión 0.8.20 (del compilador de Solidity) o superior, dado que las librerías de OpenZeppelin requieren esta versión.
4. Deployar el contrato en una VM de Remix.
5. Desde el mismo IDE online podremos interactuar con el contrato deployado.

La siguiente es una captura de la compilación y ejecución de la salida del compilador, al brindarle como entrada el caso de prueba 19 de nuestro repositorio, que consiste en un contrato que permite deployar nuevos contratos clonando la implementación de un contrato ya deployado en la blockchain.

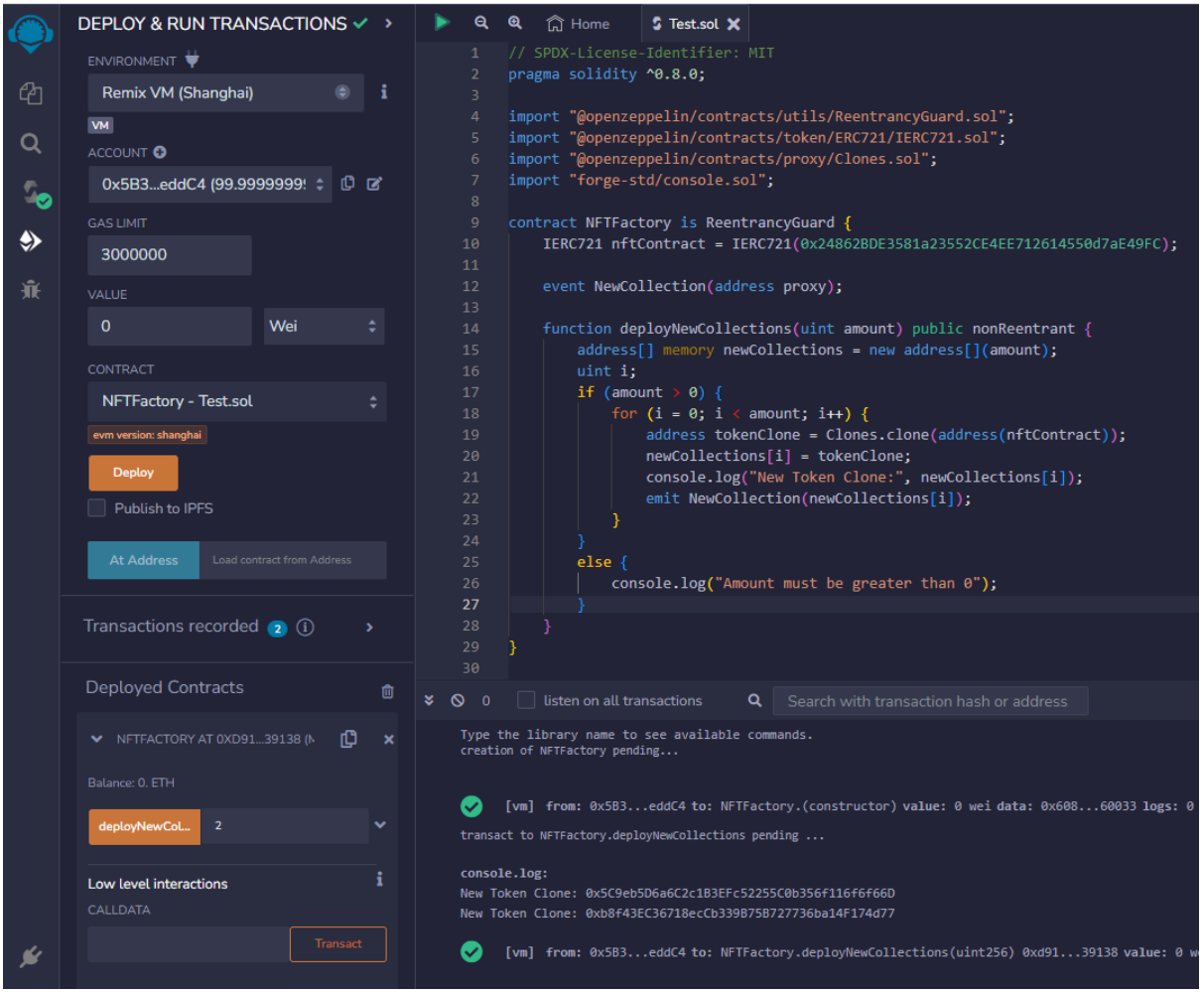

Figura 11: ejemplo de compilación y ejecución de código creado por nuestro compilador.

Foundry

Foundry es un framework de desarrollo para smart contracts en Solidity. Destaca en que todos los pasos de desarrollo, como tests y scripting se hacen en ese mismo lenguaje. Los pasos a seguir para compilar un programa son:

1. Instalar Foundry.
2. Crear un proyecto nuevo.
3. Instalar la librería de OpenZeppelin Contracts (v5.0.0).
4. Ajustar los remappings.
5. Meter el archivo de salida dentro del proyecto de Foundry, y crear un test para probarlo.

Dificultades encontradas

Las principales dificultades que nos encontramos a la hora de la implementación fueron:

- **Implementación de type checking en arrays**: inicialmente, y por la forma de las producciones, nuestra tabla de símbolos sólo indicaba para un arreglo el hecho de que era un arreglo, sin indicar su tipo. Obviamente, esto no nos permitía hacer un chequeo de tipos apropiado, por lo que tuvimos que resolverlo agregando un campo

extra a la tabla de símbolos que indique, además del tipo de dato de un identificador, a qué tipo de símbolo estaba asociado (variable, función, array o evento).
Además, por contar con un parser ascendente, tuvimos que hacer cierta iteración sobre los nodos del AST para dar con el tipo del arreglo, a la hora de construir su nodo.

- **Implementación de type checking en funciones**: igual que en el caso anterior, nos encontramos con que la información con la que disponíamos en el AST y la tabla de símbolos no era suficiente para hacer chequeos sobre los parámetros/argumentos de las funciones. Por temas de tiempo, no llegamos a solucionar esto para la entrega actual.
- **Sintaxis de Solidity**: la sintaxis de Solidity, el lenguaje al que compilamos, cuenta con reglas extremadamente complejas y estrictas. Esto nos llevó a tener que considerar diversos casos bordes durante la etapa de generación de código y chequeo de tipos para asegurarnos de producir programas válidos como salida.
- **Cantidad de gramáticas**: tratando de cubrir un amplio rango de las operaciones válidas en Solidity, terminamos con una cantidad de gramáticas considerable. Esto hizo que cada etapa de desarrollo fuera muy pesada, dado que teníamos más tipos de nodos para procesar, hacer chequeo de tipos y generación de código.
- **Lenguaje de programación**: por la simplicidad de C como lenguaje, algunas cosas requirieron escribir más código del que hubiera sido necesario en otros lenguajes. Algunas de las limitaciones que más se notaron fueron la falta de estructuras de datos nativas, y la falta de polimorfismo en las funciones.

Futuras extensiones

Las principales funcionalidades que nos gustaría agregar en un futuro al lenguaje son:

- **Valor de retorno en funciones definidas por el usuario**: esto es una funcionalidad básica que no implementamos por motivos de tiempo, e implicaría un ajuste en todas las etapas del compilador.
- **Type checking sobre parámetros/argumentos de funciones y eventos**: de nuevo, es una funcionalidad básica que no tuvimos tiempo de implementar; implicaría el guardado de información extra en la tabla de símbolos e incorporar nueva lógica al módulo de type checking.
- **Soporte para manejo de Merkle Trees y firmas ECDSA**: esta no es una extensión tan importante dado que sólo tendría sentido para un número reducido de aplicaciones, pero implicaría el agregado de nuevas funciones built-in y su correspondiente interpretación durante la etapa de generación de código.
- **Agregar decorators para manejar el control de acceso de los contratos**: actualmente no tenemos soporte nativo para control de acceso en los contratos, funcionalidad que es bastante utilizada. Esto implicaría ajustar el lexer para reconocer los nuevos decorators, y la generación de código para importar las dependencias necesarias.

- **Agregar constructores para los contratos**: implicaría ajustar todas las etapas del compilador.
- **Disponibilizar otros estándares de Ethereum como tipos de datos con métodos built-in**: implicaría ajustes en el lexer, generación de código y módulo domain-specific.

Referencias

- uthash: a hash table for C structures