Diego García Rodriguez del Campo
A01700203
18/02/2022
Diseño de Compiladores

Documentación de Ejecución de Lexer/Parser “Little Duck 2020”

Para esta tarea, se tenía que crear un compilador simple para el lenguaje “Little Duck 2020”, utilizando herramientas que podríamos usar para el proyecto de nuestro compilador.
Al tener contemplado la realización de un compilador en Rust, se utilizaron dos herramientas implementadas en Rust, que tienen su propia manera de escanear archivos de texto y determinar si la gramática es correcta o no. De igual manera, cada uno tiene su manera de crear ASTs, aunque eso no era parte de esta tarea. En este documento, documentaré la ejecución de cada herramienta, mostrando brevemente cómo se implementó, su ejecución, y explicaré lo que a mi parecer fueron las ventajas y desventajas de cada herramienta. Todos los códigos documentados en este documento se pueden encontrar en: https://github.com/diegogrdc/Little-Duck-2020

LALRPOP

LALRPOP (https://lalrpop.github.io/lalrpop/index.html) es una herramienta que nos ayuda a parsear cualquier texto dentro de Rust. Una de sus ventajas es que la sintaxis es similar a Rust, tiene un analizador léxico default, por lo que no es necesario implementar uno (puedes hacer todo con un solo Parser) aunque es posible hacerlo en caso de ser necesario. Cada regla gramatical puede ser pública o privada. En caso de ser pública, se crea un método para hacer el parse de esa regla específicamente, lo que hace muy fácil probar cada elemento poco a poco, y no tener que probar todo al mismo tiempo. Sin embargo, los errores no muestran la línea de error (aunque se puede lograr con un lexer personalizado), aunque son específicos en que falló.

Para probarlo, se creó un archivo llamado tests.rs, que prueba cada elemento de la gramática. De igual manera, se puede probar cambiando el archivo “input.txt”, y corriendo el código con el comando “cargo run”.

Con el siguiente código correcto, el compilador nos muestra un mensaje de que todo estuvo correcto.

```
program testProgram;
var a, b, c : int;
{
    a = -5;
    b = 2;
    if ((12 * (5 + 3) / 23 + 40) > b + a) {
        b = a - b;
    };
    print("The result is", a + b);
}
```

```
   Compiling lalrpop v0.19.7
   Compiling littleduck v0.1.0 (/Users/diegogarciarodriguezdelcampo/Desktop/Litt
le-Duck-2020/lalrpop)
    Finished dev [unoptimized + debuginfo] target(s) in 48.56s
     Running `target/debug/littleduck`

    Program was correctly parsed! It is written in LittleDuck2020!
```

Ahora, para probar que detecta los errores, podemos hacer algo invalido, como no escribir la primera línea “program testProgram” que es necesaria en este lenguaje. Así que borramos la línea, y obtenemos la siguiente respuesta

```
Problem parsing LittleDuck2020 file:

    UnrecognizedToken { token: (0, Token(23, "var"), 3), expected: ["\"program\""] }
```

Como vemos, se reconoció el error. Nos dice que existe un “Unrecognized token”, y que en vez de “var”, se esperaba “program”. Es muy útil, pues al momento podemos saber que falló, sin embargo, no nos dice exactamente dónde se encontró este error.
Ahora, si agregaramos la línea, pero olvidamos el punto y coma y corremos el compilador, obtenemos lo siguiente:

```
UnrecognizedToken { token: (20, Token(23, "var"), 23), expected: ["\")\"", "\"*\"", "\"+\",
```

Como vemos, se detectó el error, pero no es tan simple ver cuál es este error, pues al existir varias posibilidades, la herramienta no nos indica claramente que cambiar.

Con esta pequeña prueba, podemos ver un poco el comportamiento de esta herramienta.

grmtools

grmtools (https://softdevteam.github.io/grmtools/master/book/index.html) es una herramienta que nos ayuda a parsear cualquier texto dentro de Rust. Sin embargo, a diferencia de LALRPOP, esta herramienta si necesita un lexer y un parser. La buena noticia, es que esta herramienta está basada en Lex&Yacc, por lo que la sintaxis es muy similar y fácil de incorporar a Rust. En sus ventajas, es que los errores que se muestran son mucho más específicos e incluso te muestran algunos pasos que se pueden seguir para corregirlo. Los errores tienen líneas y se entienden bien. La sintaxis es fácil de entender, pero una de las desventajas es que la documentación es un poco corta, por lo que a veces tienes que resolver tú solo los problemas que van saliendo al implementar las cosas. Además, las pruebas que se tienen que hacer son generales, pues la gramática funciona de manera general, y no por módulos.

Para probarlo, se utilizó el archivo “input.txt”, en dónde se leía el programa y se determinaba si era válido. De igual manera, se puede probar cambiando el archivo “input.txt”, y corriendo el código con el comando “cargo run”.

Con el siguiente código correcto, el compilador nos muestra un mensaje de que todo estuvo correcto.

```
program testPrograming;
var a, b, c : int;
{
    a = -5;
    b = 2;
    if ((12 * (5 + 3) / 23 + 40) > b + a) {
        b = a - b;
    };
    print("The result is", a + b);
}
```

```
Finished dev [unoptimized + debuginfo] target(s) in 54.40s
 Running `target/debug/littleduck`

Program was correctly parsed! It is written in LittleDuck2020!
```

Ahora, para probar que detecta los errores, podemos hacer algo invalido, como no escribir la primera línea “program testProgram” que es necesaria en este lenguaje. Así que borramos la línea, y obtenemos la siguiente respuesta:

```
Parsing error at line 1 column 1. Repair sequences found:
   1: Insert PROGRAM, Insert ID, Insert SEMICOLON
```

Podemos ver que el error es muy claro. Nos dice en qué línea, y que hacer para solucionarlo. Si seguimos estas reglas, lograremos arreglar este problema en el código. Regresemos esta línea, e intentemos generar otro error, al no especificar el tipo de las variables, y solo escribir “var a, b, c;

```
Parsing error at line 2 column 12. Repair sequences found:
   1: Insert COLON, Insert FLOAT
   2: Insert COLON, Insert INT
```

Detectó el error perfectamente, e incluso nos da las opciones de lo que podemos hacer. Vemos la ventaja de esta herramienta, con su recuperación de errores.

¿Cuál es mejor?

Después de explorar ambas herramientas, podemos ver que cada una tiene sus ventajas y desventajas. Es fácil ver que los errores se detectan y se muestran mucho mejor con grmtools. Sin embargo, las pruebas son mucho más fáciles en lalrpop. La documentación de lalrpop es mejor, pero al estar basado en Yacc, podría no ser estrictamente necesario tener mucha documentación. En términos de sintaxis, ambos son muy similares, y su alcance es muy parecido. Por el momento, me inclinaría más por grmtools, pero tendría que ver la implementación de ASTs para elegir uno.