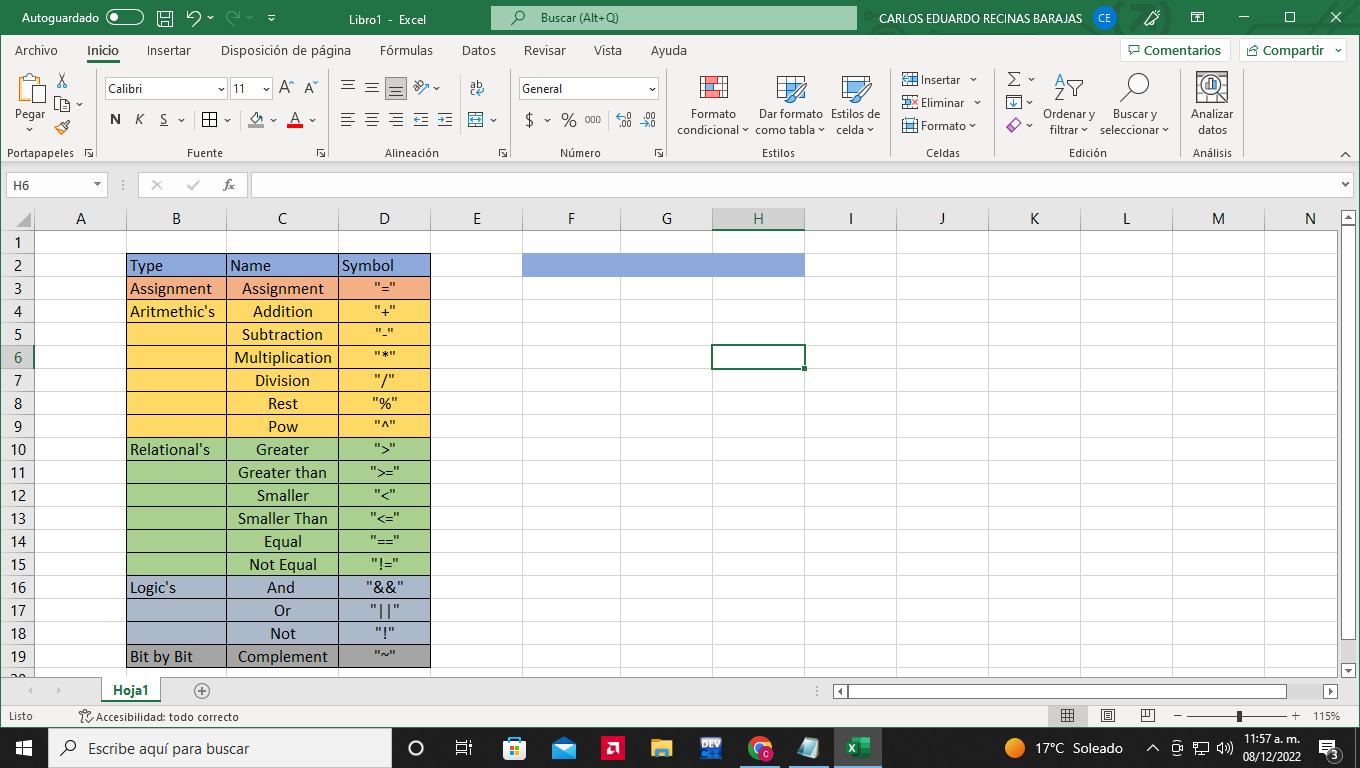
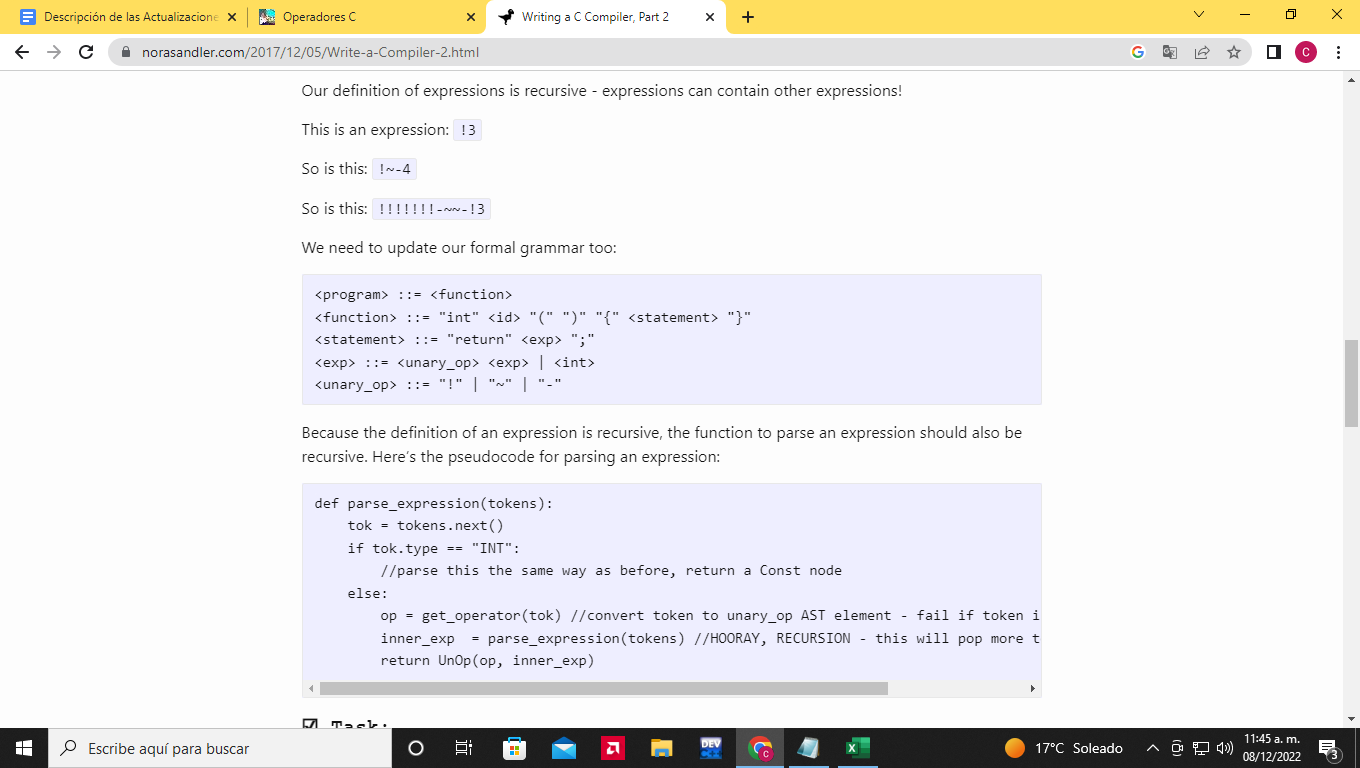
**Introducción**

En el lenguaje C las funciones retornan valores a partir de una expresión unaria, la cual mediante la combinación de un algoritmo predefinido para un operador y un sujeto llamado operando se decidirá cuál será el valor a retornar.

En particular en el lenguaje antes mencionado las expresiones tienen operadores de tipo:



Según Nora sandler creadora del artículo de “Writing a C compiler” los operadores unarios en C deben estar ordenados según la siguiente expresión regular::



Se debe tener la palabra reservada de retorno precedida de un espacio, un operando a la derecha de un operador de tipo;

* Negativo(-): El operador menos cambia el signo de su argumento. Un número positivo se vuelve negativo y un número negativo se vuelve positivo.
* Lógico(!): Se utiliza para invertir el estado lógico de su operando. Si una condición es verdadera, entonces este operador lógico la convertirá en falsa.
* Complemento(~): Convierte un valor a su forma binaria y después se cambian los valores de 0 por 1 y 1 por 0 para finalmente volver a convertir la nueva expresión a su forma decimal.

para que la expresión sea válida.

**Descripción de los cambios**

* **Argüello León Dante Moisés (PM)**

Con respecto al generador del código, se agregaron tres funciones para los operadores unarios, que se utilizarán al momento de ser leídos sus respectivos tokens al hacerle recorrido post-orden al AST. Considerando que la lectura de los operadores unarios estaría entre el token return y el token de un entero (:constant), fue necesario considerar la construcción recursiva del código desde que se llama la función emit\_code para el token :return, pues en la entrega de la Week 1, éste sólo movía el valor (:constant) al registro eax y terminaba en ret.

Ahora no solo se mueve (:constant) al registro eax, sino que considera el fragmento de código que generaría un operador unario o la anidación de estos, y termina con la vinculación de un operador unario a la función emit\_code asociada al token :constant que representa el valor entero que será operado, pues ese :constant con o sin operaciones unarias precedentes tendrá un valor final que será el que se mueva al registro rax en este caso.

Así que primero tenemos la generación del código en el caso de un :return, dónde está la instrucción mov, que será aplicada sobre el código o valor resultante después de las operaciones unarias.

def emit\_code(:return, code\_snippet, \_) do

"""

mov #{code\_snippet}

ret

"""

end

El #{code\_snippet} será el código que representa 1 o muchos operadores unarios aplicados a un :constant, o un solo valor :constant, en el caso de algún operador unario, tenemos estos nuevos códigos:

def emit\_code(:negative, code\_snippet, \_) do

"""

#{code\_snippet}

neg %rax

"""

end

def emit\_code(:complement, code\_snippet, \_) do

"""

#{code\_snippet}

not %rax

"""

end

def emit\_code(:logic, code\_snippet, \_) do

"""

#{code\_snippet}

cmp $0, %rax

mov $0, %rax

sete %al

"""

end

Y el con el token :constant será la última llamada recursiva a la función emit\_code(), de tal manera que ya en código ensamblador se posicionará el valor (value) justo enfrente de la función “mov” del emit\_code() asociado al token :return, se observa que considerando esa completitud de la función mov, se coloca el registro rax. Posteriormente, se tendrán todas las operaciones unarias si es que hay, y se llegará al “ret” que se tiene en el emit\_code() del :return.

def emit\_code(:constant, \_, value) do

"$#{value}, %rax"

end

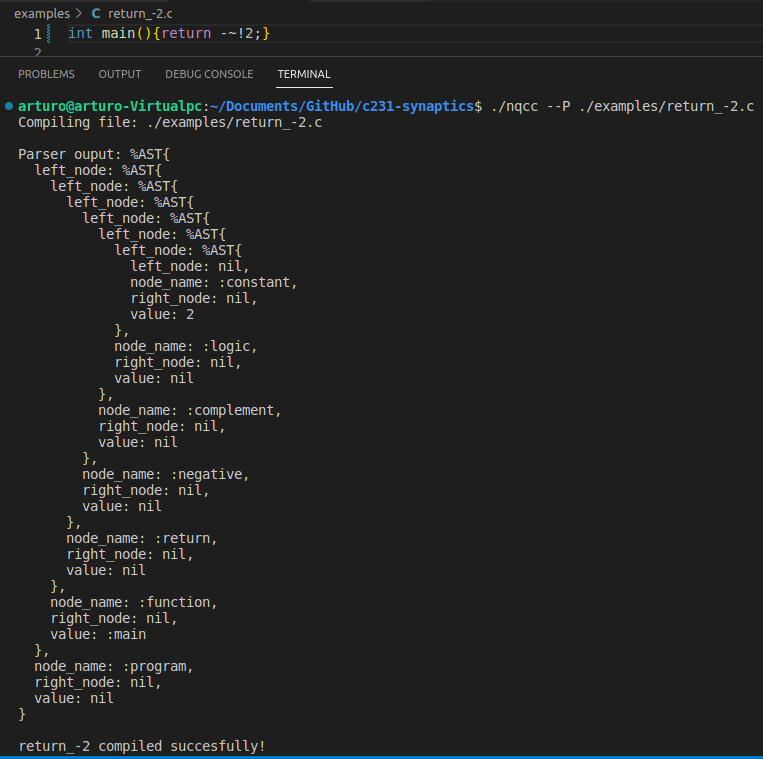
Otro cambio importante fue el uso de los registros rax, en vez de eax, esto con fin de orientarse a sistemas de 64 bits.

* **Carreto Velázquez Miguel Ángel (System Architect)**

Se modificó el parser, para generar nodos relacionados a las operaciones unitarias: negación lógica, negativo y el operador a nivel de bits (bitwise), para ello se utilizó recursividad para detectar múltiples operadores

* **Pérez Quintana Arturo (System Integrator)**

Se modificó el archivo *config.ex* para eliminar el warning que producía *Mix.Config*, la solución fue utilizar *import Config* que cumple la misma función en la versión más actual.

Se añadió funcionalidad al Parser para crear un nodo en el árbol AST por cada operador unario, para ello se modificó *parse\_expression* que antes sólo reconocía constantes y ahora es capaz de identificar tokens, sea constante, negación lógica, negativo o complemento a bit implementando recursividad llamando a la misma función *parse\_expresion* hasta que encuentre el token de constante. De esta forma el parser genera un árbol con solo ramas izquierdas que puede crecer en profundidad sin límite aparente en función de cuántos operadores unarios tenga asociada la constante, donde el nodo más profundo es la constante seguido de los operadores que se leen de derecha a izquierda hasta encontrar el *return*.

* **Recinas Barajas Carlos Eduardo (Programmer)**

Se nombró a las atómicas Negative(-), Complement(~) y Logic(!) para definir a los operadores y se desarrollaron funciones para resolver operaciones unarias dentro del Lexer, sin embargo, se toman en cuenta como actualizaciones en fase de desarrollo ya que la forma de implementar las operaciones unarias para esta entrega será distinta pero se conservarán como base para futuras entregas donde la operación tenga que ser resuelta y no se tenga que modificar tanto lo ya hecho para hacer cosas nuevas.

* ***Robledo Aguirre Eduardo (Tester)***

Fueron agregadas 30 pruebas más, 15 pruebas nuevas para el módulo “*Lexer*” y 15 para el módulo “Parser”.

Para probar el Lexer, se hace uso del archivo *lexer\_test.exs*, el cual al utilizar el comando *mix test* en la terminal, ejecuta todas las pruebas declaradas dentro de él, de forma automática e informa al usuario cuantas fueron correctas y cuales fallaron. Para probar de forma eficiente que el resultado obtenido por el lexer es correcto, se declara una lista de tokens genérica, la cual contiene los tentativos tokens que el lexer debería identificar. Para la week 1, solo se tenía una lista de tokens tentativos del siguiente código:

*int main () {*

*return 2;*

*}*

Dependiendo de las pruebas esta lista era modificada. Para esta week 2 el lexer debe tokenizar tres caracteres nuevos: Negación (-), Negación lógica (!), y Complemento binario (~). Para eficientar las pruebas automáticas, se definieron tres nuevas listas de tokens tentativos. Una lista con el operador unario negación, otra lista con el operador negación lógica y la última con el operador unario complemento binario. Además de la lista de tokens que ya se tenía de la week 1, para las pruebas definidas en esa entrega.

Dichas listas de tokens de la week 2 corresponden a los siguientes códigos:

*int main () {*

*return -2;*

*{*

*int main () {*

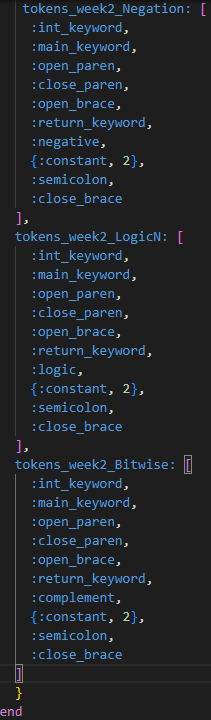
*return !2;*

*{*

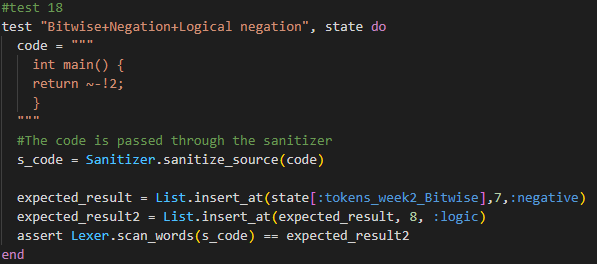
*int main () {*

*return ~2;*

*{*

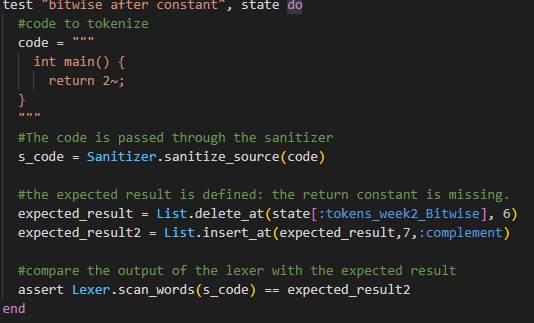


Las 15 pruebas agregadas al lexer giran en torno a estas nuevas tres listas de tokens, algunas pruebas la constante de retorno es modificada, pero sobre todo se verifica que el lexer identifique más de uno de estos operadores unarios contenidos en un mismo programa. Un ejemplo de ello es la siguiente prueba:



En dicha prueba se toma la lista de tokens con la operación unaria complemento binario y además se le agregan los operadores negación y negación lógica. Para que la prueba sea correcta, el lexer debe tokenizar esos tres operadores en el orden correcto en que aparecen en el código establecido.

Recordemos que el trabajo del lexer es solo tokenizar los caracteres que identifique, pero no revisar que estos estén en el orden correcto, tal como se muestra en la siguiente prueba:

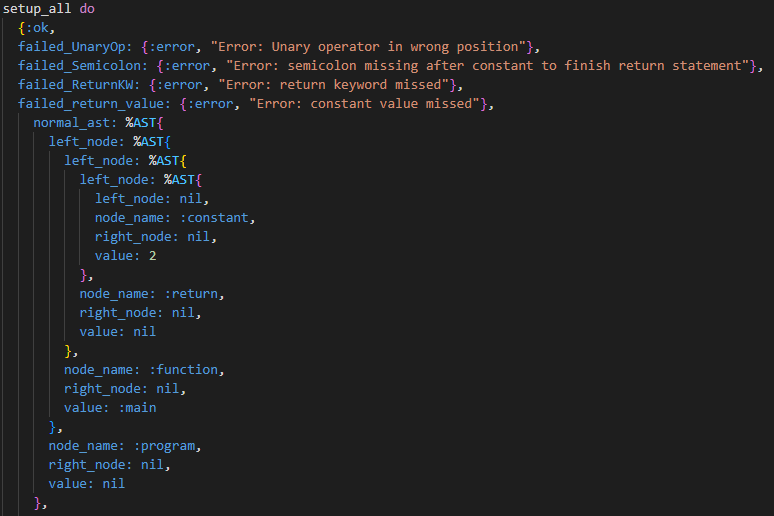
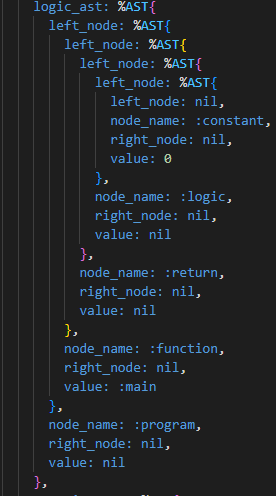


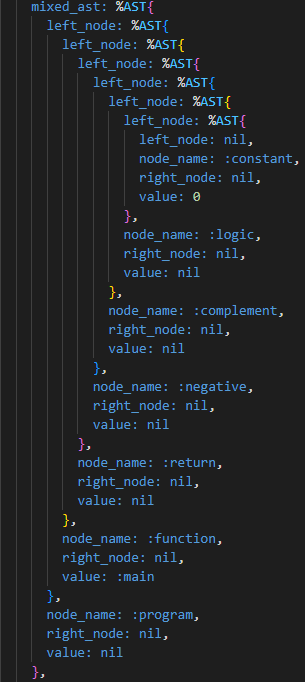
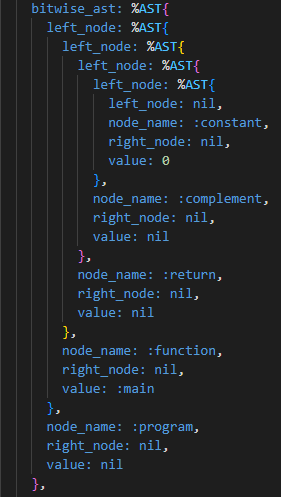
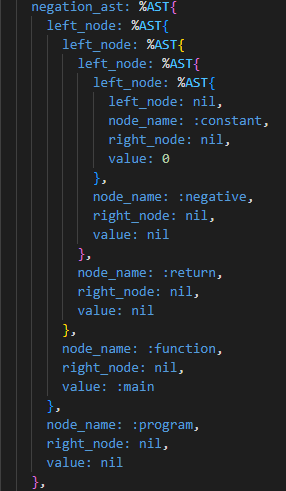
El lexer cumple con su trabajo, pues identifica y tokeniza el operador “~” que se encuentra después de la constante. Es trabajo del parser recibir la lista de tokens y darse cuenta que este operador está en la posición incorrecta, para posteriormente mandar un mensaje de error.

En el caso del parser, en esta week 2 se consideró que ya era momento de realizarle pruebas a este módulo, por lo tanto, en la carpeta test del proyecto, se creó el archivo *parser\_test.exs* y se definió que módulo sería probado en este archivo de tests.



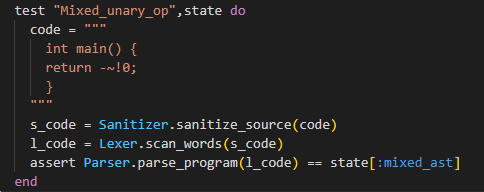
De forma similar al lexer, se definieron seis AST: “normal\_ast”, “logic\_ast”, “negation\_ast”, “bitwise\_ast”, “mixed\_ast” (refiriendose a que contiene los tres operadores unarios), y “multi\_mixed\_ast”, este último contiene múltiples operadores unarios de los tres tipos, en un orden aleatorio. Además se definieron casos de error, y su respectivo mensaje que se espera visualizar.





Cada uno de estos seis AST se utilizó para una prueba del parser. Dado que modificar un AST mediante métodos es muy complicado, las pruebas se limitaron a que dada una lista de tokens arrojada por el lexer, el parser genere el correcto AST que coincida con su correspondiente árbol de los seis definidos.

Por ejemplo, en el caso de la prueba número cinco, donde se hace uso del AST “mixed\_ast”. Se declara cierto código que contiene los tres operadores unarios, este código es pasado a través del sanitizador y posteriormente del lexer, por último y como se comentó antes, la salida del lexer se le da al parser y el AST generado es comparado con el árbol “mixed\_ast”.



Cabe mencionar que la salida del lexer, dado el código definido, es correcta, pues el mismo código de cada prueba del parser, se utilizó en pruebas del lexer y se verificó que los tokens identificados sean correctos.

En el caso de las pruebas que deben ser inválidas, se le dio al lexer una lista de tokens, que si bien contiene tokens reconocidos por el lexer, algunos de ellos están en posiciones erróneas, tal es el caso del siguiente código:

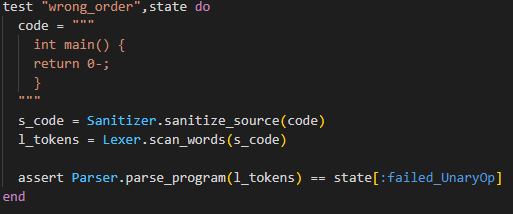
*int main () {*

*return 2-;*

*{*

Donde el operador unario negación se encuentra después de la constante de retorno y por ende la lista de tokens generada, el token del operador está después del token de la constante.

Para estas pruebas se comparó la salida del parser, que debería ser error con su respectivo mensaje, con los casos de falla definidos en el documento de pruebas. Por ejemplo:



**Conclusiones**

**Dante Argüello**

Con respecto a esta entrega, me queda más claro que la forma en cómo se está desarrollado el código requiere de conocer puntualmente dónde se realizarán los cambios, y que de cierta manera es un poco sencillo saber dónde se van a hacer los cambios debido al paradigma de la programación funcional, pues podemos identificar qué función es la que estará procesando el código a compilar en cierto punto. Entonces, identifico que cuando se requiera aumentar la complejidad de las operaciones a compilar, debemos ir primero al lexer a actualizar los tokens que podrá leer, posteriormente ir al parser, a actualizar la gramática, y finalmente ir al generador de código a crear las funciones que correspondan al código ensamblador de las operaciones que se realizarán. También observé en mayor medida la presencia de los algoritmos recursivos para la compilación, sin embargo, aún no me queda claro si para todos los compiladores en general o su gran mayoría usen la recursión, pero al menos en este proyecto sí lo veo indispensable. Otro detalle que me pareció interesante es que el generador de código debe mantener una lógica de creación correspondiente a las propiedades recursivas del algoritmo de recorrido de postorden al AST, y es necesario poder identificar en qué punto pueden o no aparecer los tokens, manteniendo las reglas de la gramática.

***Eduardo Robledo***

Con la elaboración de esta entrega, personalmente me quedo más claro el funcionamiento del parser, cómo implementar gramáticas, pensar en los posibles errores que puedan surgir y preverlos con mensajes de error, y sobre todo, como poseo el rol de tester; cómo probar el parser utilizando estos mensajes de error y los AST que sabemos son correctos. El lexer es relativamente sencillo de modificar para que reconozca los nuevos caracteres u operadores que deseemos. En el caso del generador de código no he ahondado mucho, puesto que dada la estrategia de trabajo y alcance de las pruebas por week que estableció el equipo, no he elaborado pruebas para este bloque y únicamente los programadores que se han encargado de modificarlo saben con más claridad cómo es que funciona. Sin embargo en la siguiente week debo de estudiarlo y generar las pruebas que considere pertinentes para asegurar que funcione en diversos escenarios.

**Miguel Carreto**

Sobre esta entrega entendí un poco más sobre la generación de los árboles AST y como es que se recorren. En lo personal fue un poco difícil implementar el algoritmo para detectar varios operadores más que nada por el lenguaje. Sobre aspectos de entrega aprendí un poco más sobre como hacer una presentación. Por último respecto a las herramientas utilizadas, el control de versiones se me dificultó un poco ya que hice cambios en ramas donde no iban y tuve que aprender a realizar cambios.

**Arturo Quintana**

Para esta segunda entrega fue necesario realizar una análisis más detallado sobre cómo el parser genera los nodos y cómo los va uniendo, permitiendo observar como trabaja con algoritmo descendente recursivo, de igual manera la llamada a la función parse expresion donde se realizo el principal cambio, en un inicio se me dificultó comprender en qué parte se debía implementar la recursividad para cuando existen varios operadores unarios, el observar el código previo me despejó algunas dudas sobre como ir tomando los tokens de la lista que pasa el lexer según se van ocupando pues en un inicio tomaba dos tokens y perdía el valor de alguno, para esto debía comprender cómo y cuándo extraer elementos de la cabeza de la lista y pasar el resto de manera correcta para los siguientes pasos.

Otro aspecto que se me complicó fue el generador de código ya que este debía funcionar sin importar el orden y la cantidad en el que aparecían los operadores unarios y visualizar esa posibilidad en lenguaje ensamblador no lo tenía tan claro. En cuanto a la organización del proyecto pude notar la utilidad de las ramas para hacer cambios o pruebas en el código teniendo un punto seguro al cual regresar en caso de que se tenga problemas, esto ayuda en el control de versiones por la facilidad de observar qué cambios se han hecho de una entrega a otra.

**Carlos Recinas**

Personalmente me quedo con la idea de que el software hay que trabajarlo a futuro para que sea actualizable y dejar la base para implementar esas actualizaciones a futuro y no tener que rehacer todo lo que ya se tiene por que eso anule en automático todo el tiempo ya invertido en implementaciones. Además pienso que hay que ser correcto a la hora de programar para que otros puedan entender tanto tu código, como uno entender el de otros, mediante las documentaciones u otros formas de representar y hacer más fácil la abstracción del conocimiento.