Documentación de Cambios

Compilador de Triángulo Extendido

V. 1.0.

Ramírez Ortega Angelo

2017080055

D'Ambrosio Soza Guilliano

2017158561

Alvarado flores Jose Pablo

2017127861

Bonilla Morales Ricardo

2017094182

Escuela de Ingeniería en Computación

Compiladores e Intérpretes

Profesor: Ignacio Trejos Zelaya

Instituto Tecnológico de Costa Rica

Esquema para el manejo del texto fuente:

El esquema del manejo del texto fuente no fue modificado, puesto que no requerimos cambiar ninguna funcionalidad para poder cumplir con los objetivos planteados para el proyecto.

Modificaciones hechas al analizador de léxico

Las modificaciones realizadas en el analizador léxico fueron:

- En el método scanToken se añade la funcionalidad necesaria para poder implementar los nuevos tokens:
    - Para el case donde se lee un ‘ . ‘, se definió una condición donde sí al hacer ejecutar takeIt, el currenChar es un ‘ . ’, se retorna al token de DOUBLEDOT, sino, se retorna a DOT.
    - Para el case donde se lee un ‘ : ’, se expandió la condición que venía en el caso, donde previamente se tenía la condición para el ‘=’ para retornar un BECOMES y si no lo era retornaba COLON directamente, ahora se tiene un else if donde si el carácter que le sigue es un ‘:’ se entra a otra condición, el cual pregunta, si al ejecutar takeIt el currentChar es un ‘=’ se retorna el SINGLEDECLARATION.
    - Se añade el case para el token PIPE (|)
    - Se añade el case para el token DOLLAR ($)
- Además los otros cambios que se hicieron a esta clase son los referentes a la implementación de la creación de los archivos HTML, los cambios referentes a esta funcionalidad son explicados en la siguiente sección.

Cambios hechos a los tokens y a algunas estructuras de datos

En la clase Token se le añade funcionalidad con implementación de los enteros para:

- Palabras reservadas:
    - CHOOSE
    - FOR
    - FROM
    - LOOP
    - PACKAGE
    - PAR

    - PASS
    - PRIVATE
    - RECURSIVE
    - TO
    - TYPE
    - UNTIL
    - WHEN
- Puntuaciones:
    - PIPE
    - SINGLEDECLARATION
    - DOLLAR
    - DOUBLEDOT

Y se expande tokenTable con la representación de los valores explicados anteriormente, que son representados respectivamente por:

- Palabras reservadas:
    - choose
    - for
    - from
    - loop
    - package
    - par
    - pass
    - private
    - recursive
    - to
    - type
    - until
    - when
- Puntuaciones:
    - |
    - ::=
    - $
    - ..

Explicación del esquema creado para generar la versión HTML del texto del programa fuente.

Para la generación del HTML y el manejo de los comentarios, tabulaciones, espacios en blanco, retornos de carro y cambios de línea se trabajó en el scanner. Esto ya que el scanner realiza la separación de tokens y anula los comentarios así que este punto es ideal para captar tanto los tokens separados y los comentarios que más adelante como en el

parser los comentarios, tabulaciones, espacios en blanco, retornos de carro y cambios de línea son omitidos y no se puede trabajar con ellos.

Para mantener el orden de cambios de línea en el HTML se trabajó con inline tags para la misma línea y cuando se requería cambiar de líneas se usa block tags que separan el texto en una nueva línea(bloque). Se utilizó la letra tipo Courier todo el texto, sin embargo, se siguió el formato de negrita para las palabras reservadas y los colores respectivos para las demás estructuras definidos en el enunciado del proyecto.

El archivo html se genera con el mismo nombre que el archivo de texto fuente y en la misma carpeta esto para mantener cuestiones de fácil accesibilidad y consistencia con nombres de archivo. Específicamente los cambios realizados en el código fueron en los métodos scanSeparator(), en este método se implementó lo necesario para que el HTML trate adecuadamente los tabs, los espacios en blanco, los cambios de línea y los retornos de carro ya que el HTML trata todos los espacios en blanco como uno espacio aunque sea múltiples, además aquí es donde se agrega al código HTML todos los comentarios ya que es el único lugar donde se leen.

El otro método que sobre el que se trabajó fue scan(), este método es el que finalmente devuelve el token al parser, por lo tanto antes de devolverlo se analiza cuál tipo de token es para así poder determinar cuál color debe ir ese token o si simplemente debe ir en texto HTML normal.

Cambios realizados a las reglas sintácticas de 🔺 extendido (Gramática)

Con el fin de cumplir con los requerimientos que se solicitan del proyecto, se tuvo que agregar o eliminar diferentes reglas sintácticas de la gramática. Además, con el fin de que el análisis sintáctico fuera más eficiente, se eliminó la recursividad por la izquierda, y se factorizó aquellas reglas que se pudieran factorizar.

Respecto al production de single-Command se eliminaron las siguientes reglas sintácticas:

| **begin** Command **end**
| **let** Declaration **in** single-Command
| **if** Expression **then** single-Command
else single-Command
| **while** Expression **do** single-Command

En el mismo production, se agregaron las siguientes reglas sintácticas:

| **pass**
| **loop** Loop-Cases
| **let** Declaration **in** Command **end**
| **if** Expression **then** Command **else** Command **end**
| **choose** Expression **from** Cases **end**

Además se modificó la regla que decía Identifier **(**Actual-Parameter-Sequence**)** por Long-Identifier **(**Actual-Parameter-Sequence**)**, en el mismo production de single-Command.

Debido a las reglas sintácticas que se agregaron en single-Command se tuvo que agregar el production de Loop-Cases. El cual es el siguiente:

Loop-Cases ::= **while** Expression **do** Command **end**
| **until** Expression **do** Command **end**

| **do** Command Do-loop
| **for** Identifier **from** Expression **to** Expression For-loop

Del mismo modo, se agregaron los respectivos productions de Do-loop y For-loop, los cuales son:

Do-loop ::= **while** Expression **end**
| **until** Expression **end**

For-loop ::=**do** Command **end**
| **while** Expression **do** Command **end**
| **until** Expression **do** Command **end**

Estos productions de loops, permitieron factorizar los loops que se solicitaron agregar al production single-Command en el enunciado. Luego, en el enunciado del proyecto se solicitó que se agregaran los siguientes productions: Cases, Case, ElseCase, Case-Literals, Case-Range y Case-Literal. Los cuales se presentan a continuación:

Cases ::= (Case)+ [ElseCase]

Case ::= **when** Case-Literals **then** Command

ElseCase ::= **else** Command

Case-Literals ::= Case-Range (**|** Case-Range)*

Case-Range ::= Case-Literal [**..** Case-Literal]

Case-Literal ::= Integer-Literal
| Character-Literal

Respecto a las expressions de la sintaxis, se modificó el production secondary-Expression, de modo que este quedara factorizado de la siguiente manera:

secondary-Expression ::= primary-Expression (Operator primary-Expression)*

Además, en el production de primary-Expression se modificó la regla sintáctica de Identifier **(** Actual-Parameter-Sequence **)** por Long-Identifier **(** Actual-Parameter-Sequence **)**.

Asimismo, se factorizan los productions: record-aggregate y array-agregate, de modo que estos quedaron de la siguiente manera:

Record-Aggregate ::= Identifier **~** Expression [**,** Record-Aggregate]

Array-Aggregate ::= Expression [**,** Array-Aggregate]

Otra modificación que solicitaba el enunciado es la del production V-name, el cual antes de modificarse era de la siguiente manera:

V-name ::= Identifier
| V-name **.** Identifier
| V-name **[** Expression **]**

Luego de modificarse basado en el enunciado, se obtuvo el siguiente production:

V-name ::= [Package-Identifier **$**] Var-Name

Esto a su vez implicó que se tuviera que agregar el production Var-Name, pero el que se indicaba en el enunciado presentaba recursión por la izquierda, por lo que en vez del production:

Var-name ::= Identifier

| Var-name **.** Identifier

| Var-name **[** Expression **]**

Se agregó el siguiente production:

Var-name ::= Identifier (Selector)*

El production Selector permite manejar la recursión por la izquierda que presentaba el production que se solicitaba en el enunciado. Este production Selector es el siguiente:

Selector ::= **.**Identifier

| **[** Expression **]**

También, el enunciado solicitó modificar el production Declaration, de modo que este quedara de la siguiente manera:

Declaration ::= compound-Declaration

| Declaration **;** compound-Declaration

Dado que el production propuesto por el enunciado presenta recursión por la izquierda, este se redujo, y se obtuvo el siguiente:

Declaration ::= compound-Declaration (**;** compound-Declaration)*

A partir de esta modificación se obtuvo un nuevo production, el cual es compound-Declaration, el cual se presenta a continuación:

compound-Declaration ::= single-Declaration

| **recursive** Proc-Funcs **end**

| **private** Declaration **in** Declaration **end**

| **par** single-Declaration (**|** single-Declaration)$^{+}$ **end**

Del mismo modo, se tuvieron que añadir los productions Proc-Func y Proc-Funcs, estos son:

Proc-Func ::= **proc** Identifier **(** Formal-Parameter-Sequence **)**

~ Command **end**

| **func** Identifier **(** Formal-Parameter-Sequence **)**

: Type-denoter **~** Expression

Proc-Funcs ::= Proc-Func (**|** Proc-Func)+

Luego, a single-Declaration basado en el enunciado se le tuvo que modificar la regla proc Identifier **(** Formal-Parameter-Sequence **) ~** single-Command, por la regla proc Identifier **(** Formal-Parameter-Sequence **) ~** Command **end**. Del mismo modo, se solicitó agregar la regla **var** Identifier **::=** Expression, pero como había otra regla que igual comenzaba por medio de la terminal **var**, esta se factorizó, de modo que single-Declaration quedó de la siguiente manera:

single-Declaration ::= **const** Identifier **~** Expression

| **var** Identifier Var-Single-Declaration

| **proc** Identifier **(** Formal-Parameter-Sequence **)**

~ Command **end**

| **func** Identifier **(** Formal-Parameter-Sequence **)**

: Type-denoter **~** Expression

| **type** Identifier **~** Type-denoter

Al factorizar lo anterior se generó el nuevo production denominado Var-Single-Declaration, el cual es:

Var-Single-Declaration ::= **:** Type-denoter

| **::=** Expression

Al revisar la gramática del libro, se observó que el production Formal-Parameter-Sequence tenía un épsilon en una de sus reglas, y dado que el enunciado del proyecto solicitaba que estas se eliminarán, el production quedó de la siguiente manera:

Formal-Parameter-Sequence ::= [proper-Formal-Parameter-Sequence]

Del mismo modo, el production proper-Formal-Parameter-Sequence poseía dos reglas que iniciaban a través del mismo terminal, por lo que se factorizó el production, y quedó de esta manera:

proper-Formal-Parameter-Sequence ::= Formal-Parameter [**,** proper-Formal-Parameter-Sequence]

Asimismo, el production Actual-Parameter-Sequence poseía un épsilon, por lo que se reemplazó con el siguiente production:

Actual-Parameter-Sequence ::= [proper-Actual-Parameter-Sequence]

En el caso del production proper-Actual-Parameter-Sequence, dado que dos de las reglas iniciaban con el mismo no terminal, este se factorizó de la siguiente manera:

proper-Actual-Parameter-Sequence ::= Actual-Parameter [, proper-Actual-Parameter-Sequence]

Además, se solicitó modificar el production Type-denoter, de manera que la regla que este poseía de Identifier, pasará a ser Long-Identifier, por lo que se obtuvo el siguiente production:

Type-denoter ::= Long-Identifier

I **array** Integer-Literal **of** Type-denoter

I **record** Record-Type-denoter **end**

El production Record-Type-denoter pasó de ser dos reglas sintácticas que comenzaban con el mismo no terminal, a una sola, dado que se factorizó de la siguiente manera:

Record-Type-denoter ::= Identifier **:** Type-denoter[**,** Record-Type-denoter]

También, el enunciado solicitó que se agregara el production de Package-Declaration, el cual es:

Package-Declaration ::= **package** Package-Identifier **~**
Declaration **end**

Finalmente, el enunciado solicitó una serie de cambios léxicos, los cuales fueron:

- Añadir las palabras reservadas **choose, for, from, loop, par, pass, private, recursive, to, until** y **when** como nuevas alternativas en la especificación de Token.
- Añadir los símbolos **|**, **::=**, **$** y **. .** como nuevas alternativas de Token.
- Eliminar la palabra reservada **begin**.
- Los identificadores en mayúsculas son significantes y distintas de las minúsculas.

Luego de todas las modificaciones mencionadas anteriormente, la gramática final es la siguiente:

Program ::= (Package-Declaration **;**)* Command

Command ::= single-Command (**;** single-Command)*

single-Command ::= V-name **:=** Expression
| Long-Identifier **(** Actual-Parameter-Sequence **)**
| **pass**
| **loop** Loop-Cases
| **let** Declaration **in** Command **end**
| **if** Expression **then** Command **else** Command **end**
| **choose** Expression **from** Cases **end**

Loop-Cases ::= **while** Expression **do** Command **end**
| **until** Expression **do** Command **end**
| **do** Command Do-loop
| **for** Identifier **from** Expression **to** Expression For-loop

Do-loop ::= **while** Expression **end**
| **until** Expression **end**

For-loop ::=**do** Command **end**
| **while** Expression **do** Command **end**
| **until** Expression **do** Command **end**

Cases ::= (Case)+ [ElseCase]

Case ::= **when** Case-Literals **then** Command

ElseCase ::= **else** Command

Case-Literals ::= Case-Range (**|** Case-Range)*

Case-Range ::= Case-Literal [**..** Case-Literal]

Case-Literal ::= Integer-Literal
| Character-Literal

Expression ::= secondary-Expression
I **let** Declaration **in** Expression
I **if** Expression **then** Expression **else** Expression

secondary-Expression ::= primary-Expression (Operator primary-Expression)*

primary-Expression ::= Integer-Literal
I Character-Literal
I V-name
I Long-Identifier **(** Actual-Parameter-Sequence **)**

| Operator primary-Expression
| **(** Expression **)**
| **{** Record-Aggregate **}**
| **[** Array-Aggregate **]**

Record-Aggregate ::= Identifier **~** Expression [**,** Record-Aggregate]

Array-Aggregate ::= Expression [**,** Array-Aggregate]

V-name ::= [Package-Identifier **$**] Var-Name

Var-name ::= Identifier (Selector)*

Selector ::= **.**Identifier
| **[** Expression **]**

Declaration ::= compound-Declaration (**;** compound-Declaration)*

compound-Declaration ::= single-Declaration
| **recursive** Proc-Funcs **end**
| **private** Declaration **in** Declaration **end**
| **par** single-Declaration (**|** single-Declaration)$^{+}$ **end**

Proc-Func ::= **proc** Identifier **(** Formal-Parameter-Sequence **)**
~ Command **end**
| **func** Identifier **(** Formal-Parameter-Sequence **)**
: Type-denoter **~** Expression

Proc-Funcs ::= Proc-Func (**|** Proc-Func)+

single-Declaration ::= **const** Identifier **~** Expression
| **var** Identifier Var-Single-Declaration
| **proc** Identifier **(** Formal-Parameter-Sequence **)**
~ Command **end**
| **func** Identifier **(** Formal-Parameter-Sequence **)**
: Type-denoter **~** Expression
| **type** Identifier **~** Type-denoter

Var-Single-Declaration ::= **:** Type-denoter
| **::=** Expression

Formal-Parameter-Sequence ::= [proper-Formal-Parameter-Sequence]

proper-Formal-Parameter-Sequence ::= Formal-Parameter [**,**
proper-Formal-Parameter-Sequence]

Formal-Parameter ::= Identifier **:** Type-denoter
| **var** Identifier **:** Type-denoter
| **proc** Identifier **(** Formal-Parameter-Sequence **)**
| **func** Identifier **(** Formal-Parameter-Sequence **)**
: Type-denoter

Actual-Parameter-Sequence ::= [proper-Actual-Parameter-Sequence]

proper-Actual-Parameter-Sequence ::= Actual-Parameter [,
proper-Actual-Parameter-Sequence]

Actual-Parameter ::= Expression
| **var** V-name
| **proc** Identifier
| **func** Identifier

Type-denoter ::= Long-Identifier
I **array** Integer-Literal **of** Type-denoter
I **record** Record-Type-denoter **end**

Record-Type-denoter ::= Identifier **:** Type-denoter[**,** Record-Type-denoter]

Package-Declaration ::= **package** Package-Identifier **~**
Declaration **end**

Microsintaxis

Package-Identifier ::= Identifier

Long-Identifier ::= [Package-Identifier **$**] Identifier

Token ::= Integer-Literal | Character-Literal | Identifier | Operator | **array** | **choose** | **const** | **do** | **else** | **end** | **for** | **from** | **func** | **if** | **in** | **let** | **loop** | **of** | **package** | **par** | **pass** | **private** | **proc** | **record** | **recursive** | **then** | **to** | **type** | **until** | **var** | **when** | **while** | **.** | **:** | **;** | **,** | **:=** | **~** | **(** | **)** | **[** | **]** | **{** | **}** | | | **::=** | **$** | **. .**

Integer-Literal ::= Digit Digit*

Character-Literal ::= **‘** Graphic **‘**

Identifier ::= Letter (Letter | Digit)*

Operator ::= Op-character Op-character*

Comment ::= **!** Graphic* end-of-line

Blank ::= space | tab | end-of-line

Graphic ::= Letter | Digit | Op-character | space | tab | **.** | **:** | **;** | **,** | **~** | **(** | **)** | **[** | **]** | **{** | **}** | **!** | **`** | **”** | **#** | **$**

Letter ::= **a** | **b** | **c** | **d** | **e** | **f** | **g** | **h** | **i** | **j** | **k** | **l** | **m** | **n** | **o** | **p** | **q** | **r** | **s** | **t** | **u** | **v** | **w** | **x** | **y** | **z** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** | **J** | **K** | **L** | **M** | **N** | **O** | **P** | **Q** | **R** | **S** | **T** | **U** | **V** | **W** | **X** | **Y** | **Z**

Digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

Op-character ::= **+** | **-** | ***** | **/** | **=** | **<** | **>** | **** | **&** | **@** | **%** | **^** | **?**

Program ::= (Token | Comment | Blank)*

Nuevas rutinas de reconocimiento sintáctico

En esta sección se van a explicar cuáles fueron los cambios hecho en el parser con el fin de extender el reconocimiento sintáctico, se tienen dos variantes, los métodos modificados y los nuevos.

Para todos los casos de los métodos definidos a continuación, se retorna el AST correspondiente a dicho caso, es decir correspondiente a cada expresión posible en la gramática. Además los cambios hechos dentro de los métodos como por ejemplo la transformación de los identifier a compound declaration o de singleCommand a CompoundCommand, se dan por explicados en la sección de cambios en la gramática.

Las rutinas modificadas son:

- ParseProgram(): se modificó este método para poder incluir los paquetes. Primeramente se pregunta si existe el token “package”, de ser así, parsea el paquete y busca la existencia del token “;”. Si existe, se parsea otro paquete. De lo contrario, se parsea el comando. De no existir un comando, se coloca un comando vacío.
- parseSingleCommand(): se modificó este método para poder añadir los casos de Long-Identifier, pass, loops, choose y eliminar el de begin. Además se eliminan los casos de SEMICOLON, END, ELSE, IN.
- parseExpression(): no se modificó la lógica del método, sin embargo se cambió el orden de los casos y se añadieron tokens al primer de los casos para cumplir con los cambios en la gramática pero en las producciones que están por debajo de esta, es decir la expresión secundaria.
- parsePrimaryExpression(): Se modificó este método para añadir los casos de LongIdentifier y VName, como ambos empiezan con identificadores, primero se busca el identificador y, de tener adyacente un token “(”, se determina que

es una llamada y por lo tanto un long identifier. De lo contrario, se determina que es una asignación.

- parseVname(): se modificó para poder incluir los identificadores compuestos con paquetes. Se parsea un identificador y de encontrar el token “$” se determina que es un identificador de paquete y luego se busca el identificador de la variable. De lo contrario, se asigna como identificador de variable. Posteriormente se utiliza restOfVname para encontrar los selectores.
- parseRestOfVname(): se modificó este método para poder añadir los identificadores que ahora también pueden incluir un identificador de paquete.
- parseDeclaration(): se modificó de manera que ahora utiliza declaraciones compuestas en vez de únicas.
- parseFormalParameter(): Se modificó el caso de FUNC y se añadió el caso de VAR.
- parseActualParameter(): Se modificó este método para poder añadir el caso de VAR.
- parseTypeDenoter(): Se modificó este método para poder añadir los casos de LongIdentifier, y Record Type Denoter.

Las nuevas rutinas son:

- parseLongIdentifier(): se creó este método para poder parsear a los long identifier el cual, parsea el identificador en el método anterior y si el token actual es un $ de dólar crea el AST como un PackageIdentifier y sino únicamente como un LongIdentifierparseRecordTypeDenoter().
- parseLoopCases: este metodo se encarga de parsear los casos de loop donde sus posibles valores iniciales son while, until, do y for, estos correspondientemente llaman a los métodos que necesiten ya sean las expresiones, identificador o los comandos, o bien como en el caso de do, llama al método de parseDoLoop.
- parseDoLoop(): este método se encarga de manejar los posibles casos del do, donde estos posibles valores iniciales son until y while.
- parseForLoop(): este método se encarga de manejar los posibles casos del for, donde sus posibles valores iniciales son do, until y while.
- parseCases(): este método se encarga de manejar los casos en el singleCommand de choose, este define que puede ser uno o más Case y con su opcional ElseCase. Por lo que se tiene un while que crea un SequentialCase hasta que ya no haya un When como token actual, y un if que determina si se añadió el ElseCase.
- parseCase(): este método se encarga de parsear los casos individuales del Choose, el cual tiene que empezar con el token de When.
- parseElseCase(): este método se encarga de parsear el caso opcional del Choose el cual es del else.
- parseCaseLiterals(): este método se encarga de parsear los CaseLiterals los cuales están compuesto de rangos de n literales, como 1..2 o literales como ‘a’ separados por medio del token PIPE, por lo que se crea un SequentialCaseRange mientras haya un PIPE como current token.

- parseCaseRange(): este método se encarga de parsear al Case range que puede estar compuesto de un Case literal o de dos separados por un token DOUBLEDOT.
- parseCaseLiteral(): este método se encarga de parsear los literales que pueden ser un literal entero o un literal caracter.
- parseCompoundIdentifier(): se creó este método para parsear aquellos identificadores que cuentan con un paquete opcional. Primero se parsea un identificador, luego se determina si es de paquete o no, mediante la existencia del token “$”. Si es de paquete, se busca el identificador de la variable, si no, se le asigna el inicial.
- parseCompoundDeclaration(): se creó este método para poder parsear las declaraciones compuestas, aquellas declaraciones que contienen declaraciones recursivas, privadas o pares, según definidas en la gramática.
- parseVarSingleDeclaration(): se creó este método para pode parsear las declaraciones de variables. Si se tiene un “:”, entonces se espera un denotador de tipo. Si se tiene un “:=” se espera una asignación.
- parseProcFunc(): se creó este método para poder parsear las funciones y los procedimientos. Un procedimiento está compuesto por un identificador, una serie de parámetros y los comandos de ejecución. Una función está compuesta por un identificador, una serie de parámetros su tipo de retorno y una expresión
- parseProcFuncs(): se creó este método para poder parsear los procedimientos y funciones múltiples, los cuales pueden ser una secuencia de cualquiera de ambos, con al menos dos.
- parseRecordTypeDenoter(): se creó este método para poder parsear los tipos de records, los cuales pueden ser una secuencia de identificadores con sus respectivos denotadores de tipos.
- parsePackageDeclaration(): se creó este método para poder parsear las declaraciones de paquetes, las cuales están compuestas por un identificador y una serie de declaraciones.

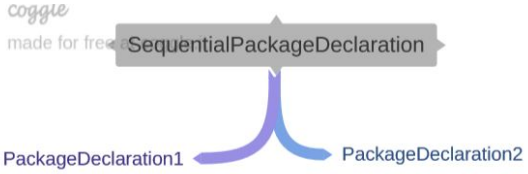
Lista de errores sintácticos detectados

No se detectaron errores sintácticos

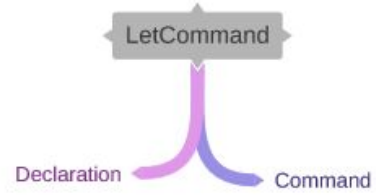
Modelaje realizado para los árboles sintácticos

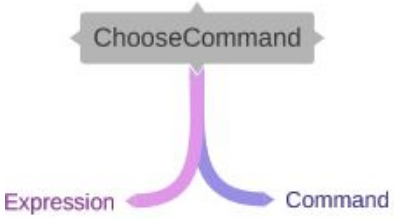
Con el fin de llevar un mayor orden en la programación lógica de los nuevos métodos del parser, y comprender cómo se crea y recorre el AST, se realizaron diferentes representaciones de los árboles sintácticos, por cada una de las categorías: program, commands, loops, cases, expressions, aggregates, value-or-variable names, declarations, procs, parameters, type denoters y literals, identifiers y operators.

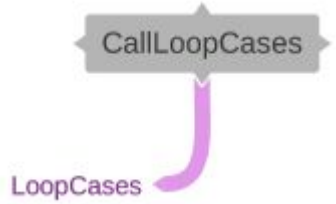
Representaciones de la Categoría Program

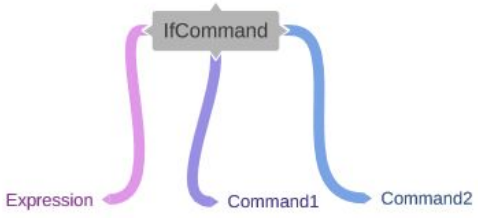


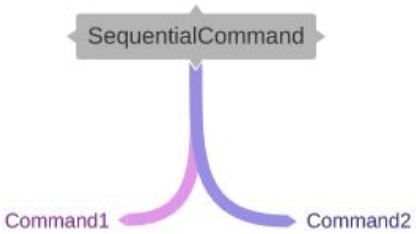
Representaciones de la Categoría Commands

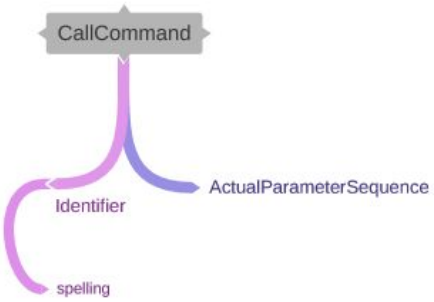




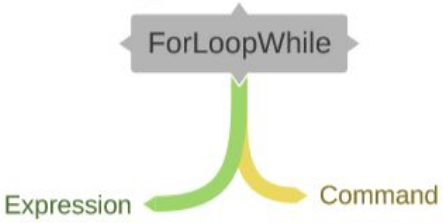


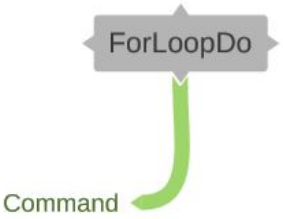


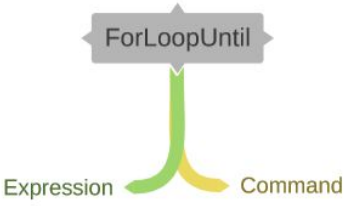


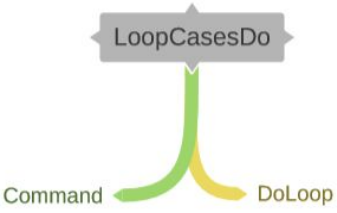


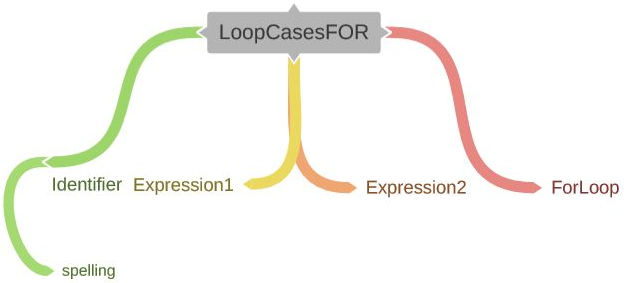
Representaciones de la Categoría Loops

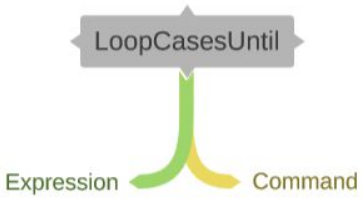


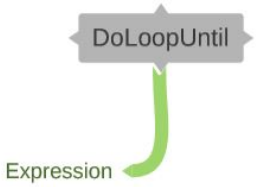


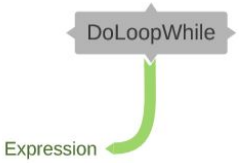


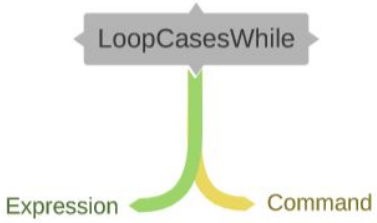




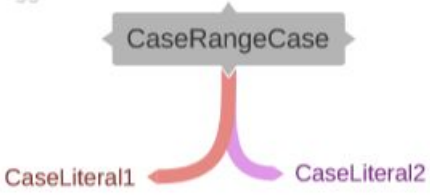


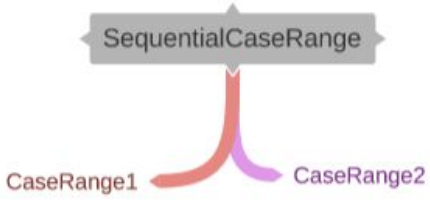


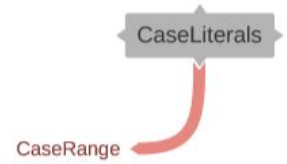


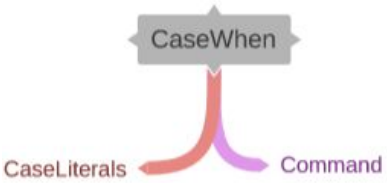


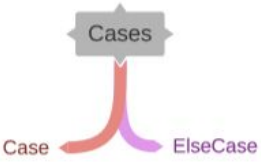
Representaciones de la Categoría Cases

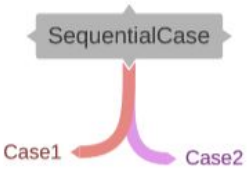


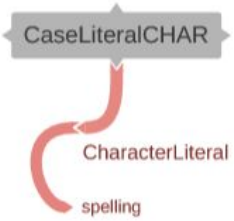


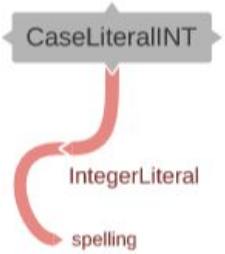


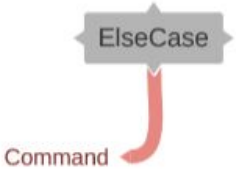




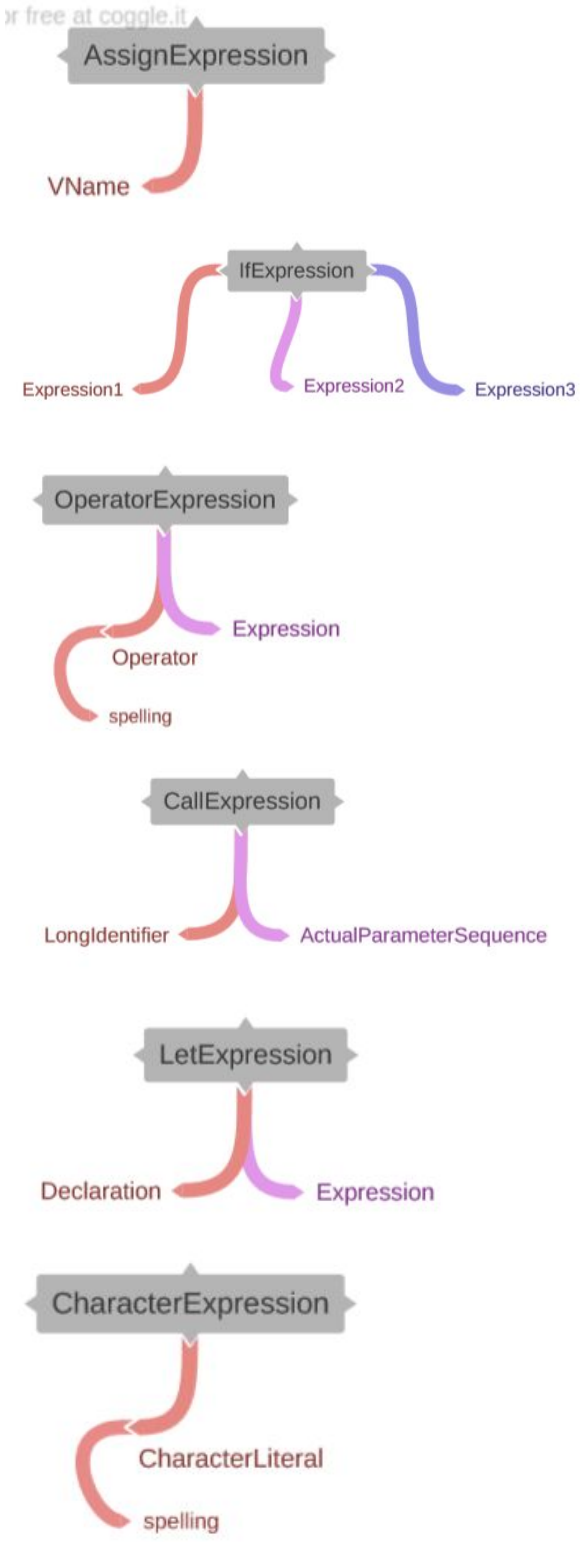


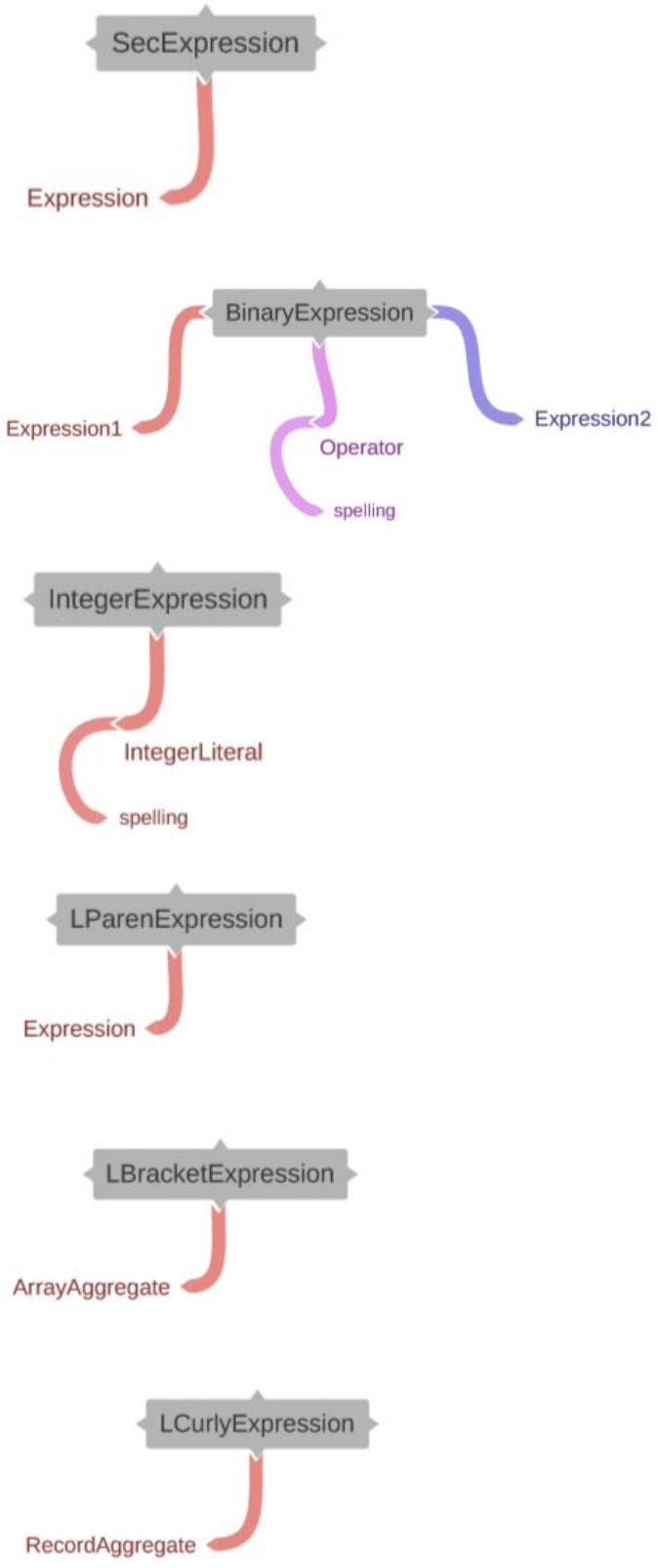




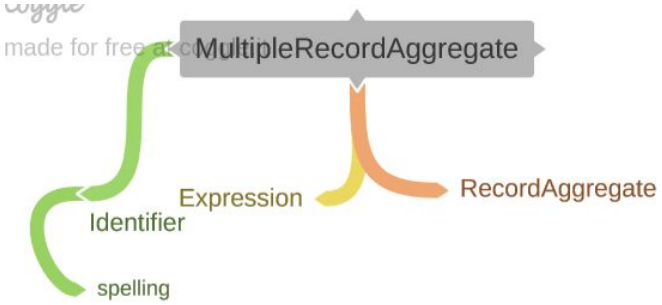


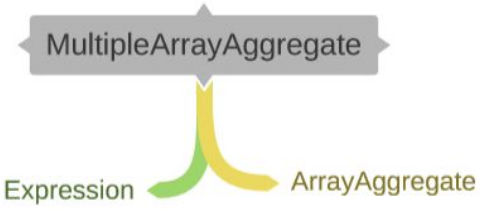
Representaciones de la Categoría Expressions

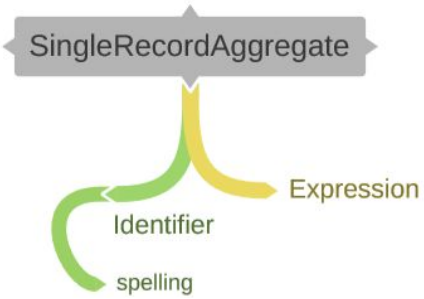


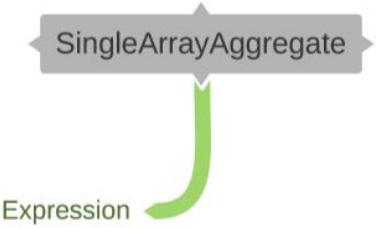


Representaciones de la Categoría Aggregates

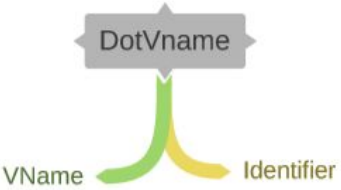


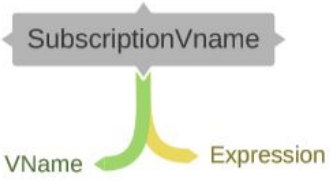


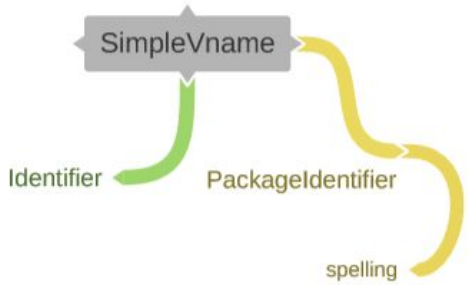


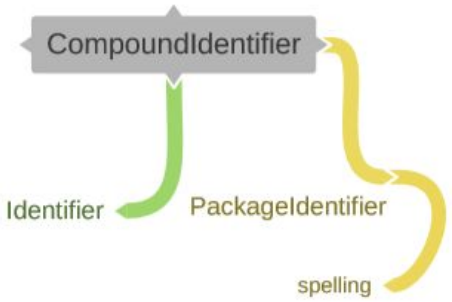


Representaciones de la Categoría Value-or-Variable Names

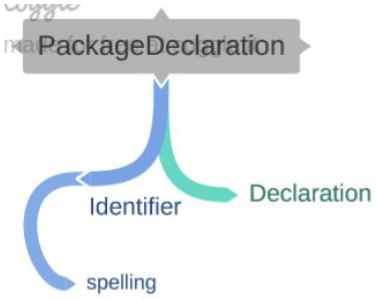


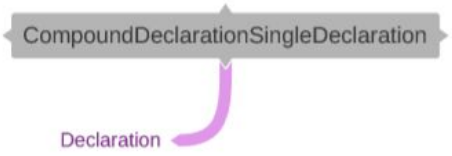


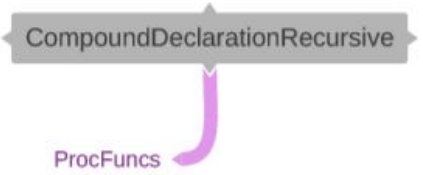


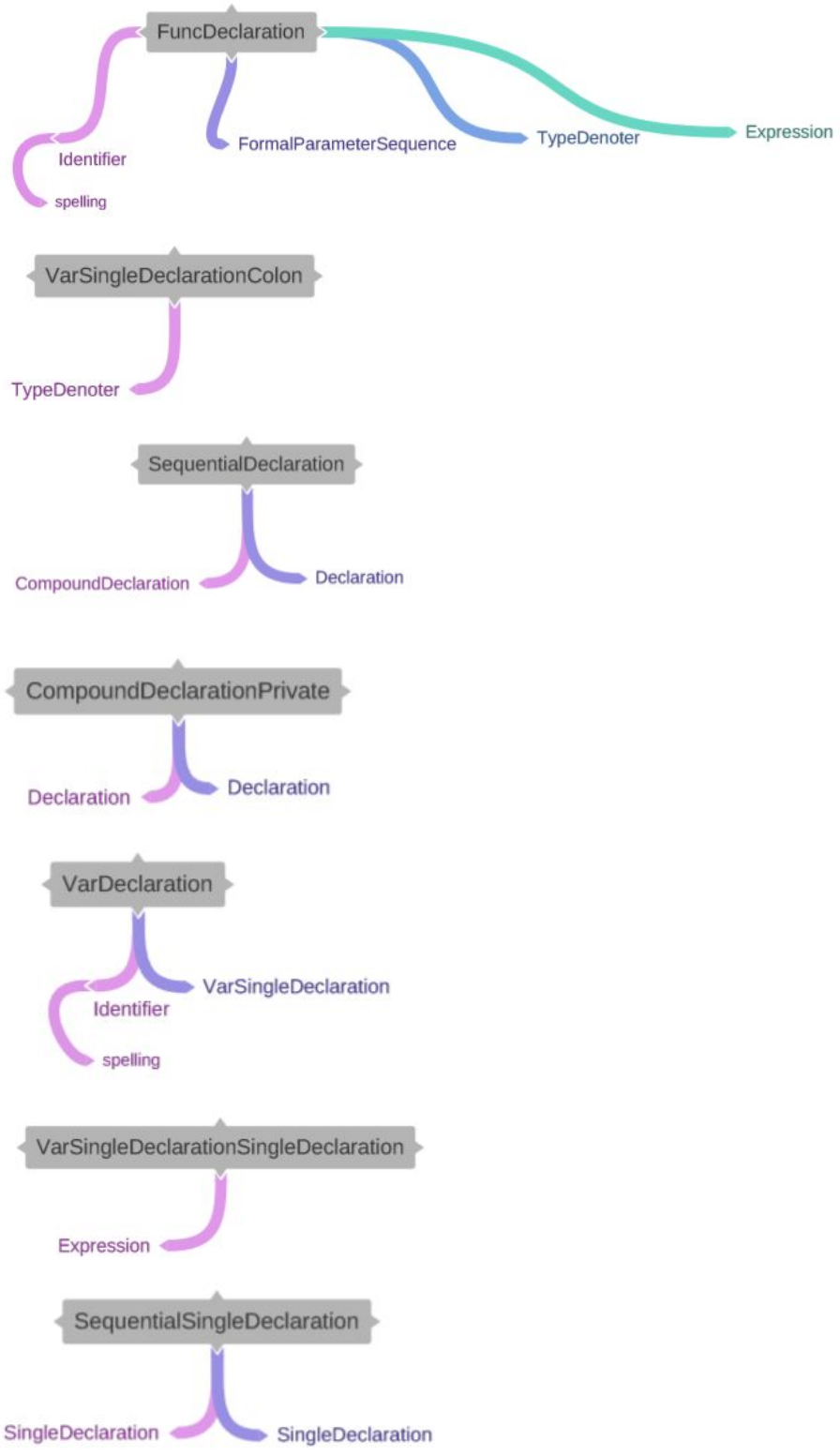


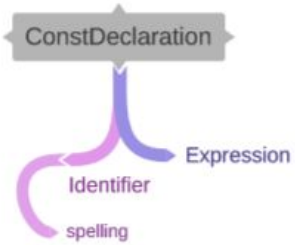
Representaciones de la Categoría Declarations

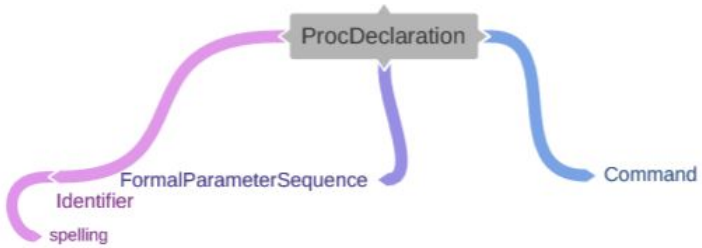


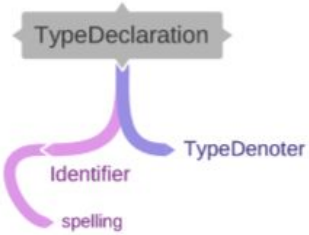




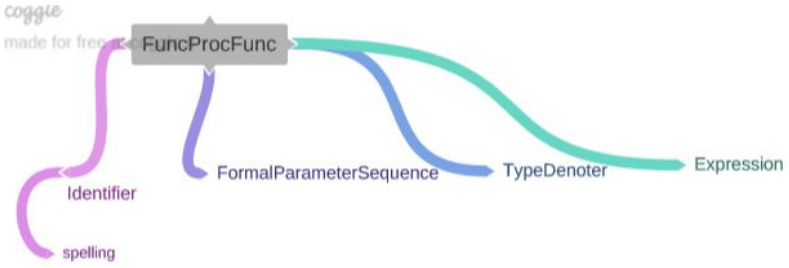

FuncDeclaration
Identifier
spelling
FormalParameterSequence
TypeDenoter
Expression
VarSingleDeclarationColon
TypeDenoter
SequentialDeclaration
CompoundDeclaration
Declaration
CompoundDeclarationPrivate
Declaration
Declaration
VarDeclaration
Identifier
spelling
VarSingleDeclaration
VarSingleDeclarationSingleDeclaration
Expression
SequentialSingleDeclaration
SingleDeclaration
SingleDeclaration

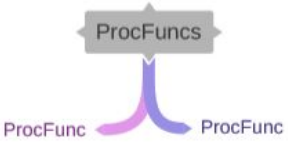


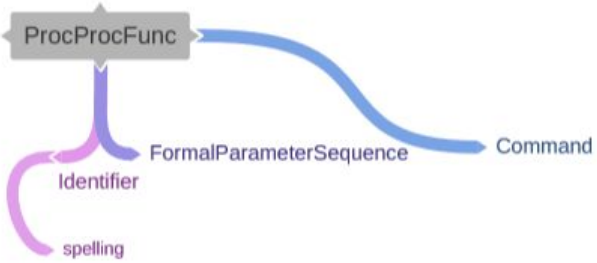




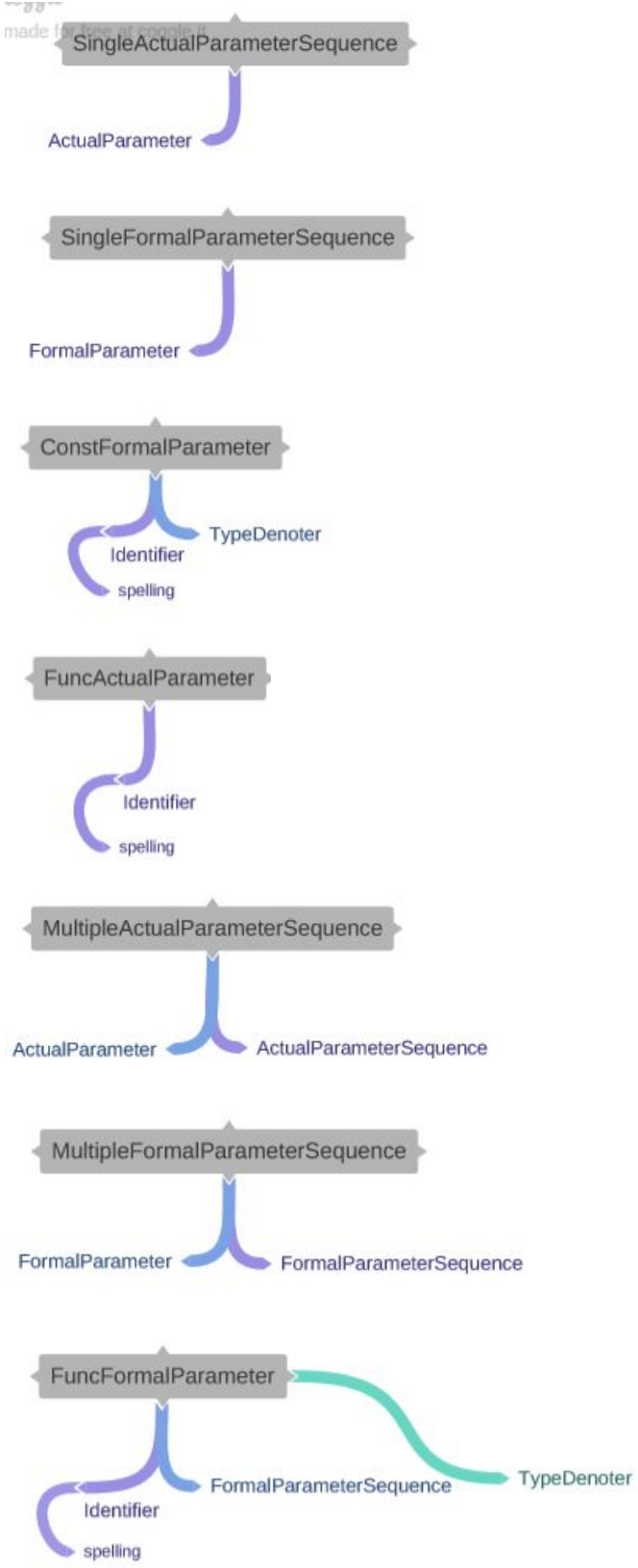
Representaciones de la Categoría Procs

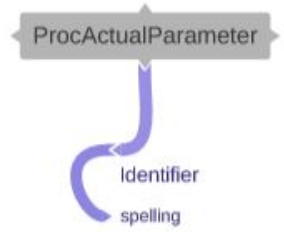


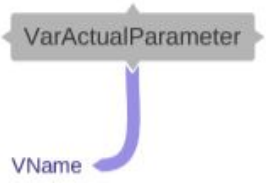


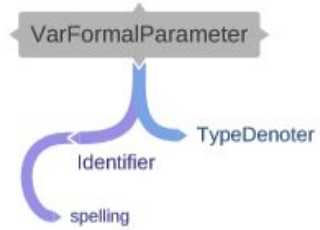


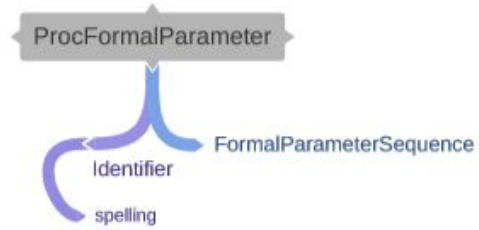
Representaciones de la Categoría Parameters

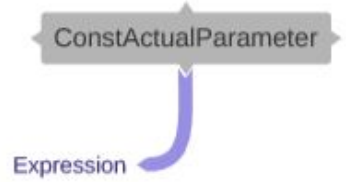

SingleActualParameterSequence
ActualParameter
SingleFormalParameterSequence
FormalParameter
ConstFormalParameter
TypeDenoter
Identifier
spelling
FuncActualParameter
Identifier
spelling
MultipleActualParameterSequence
ActualParameter
ActualParameterSequence
MultipleFormalParameterSequence
FormalParameter
FormalParameterSequence
FuncFormalParameter
FormalParameterSequence
TypeDenoter
Identifier
spelling



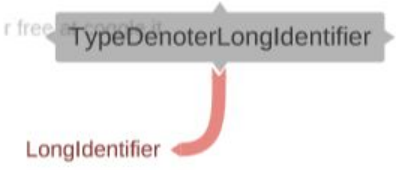


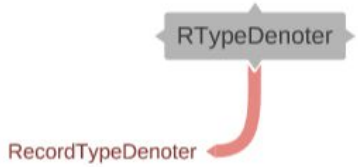


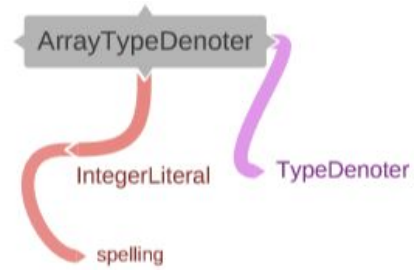


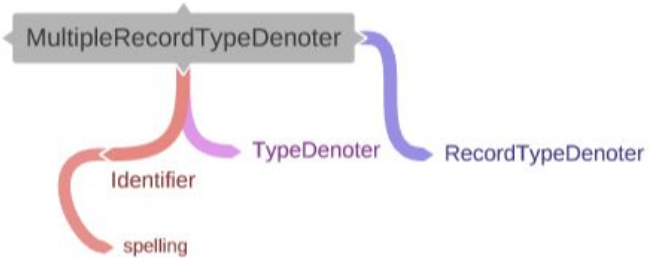


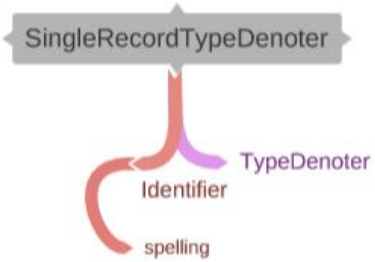
Representaciones de la Categoría Type Denoters



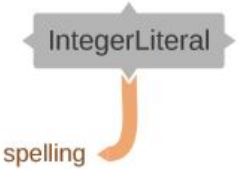


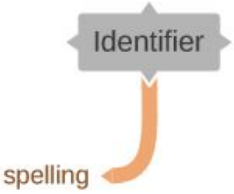


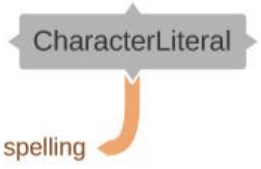


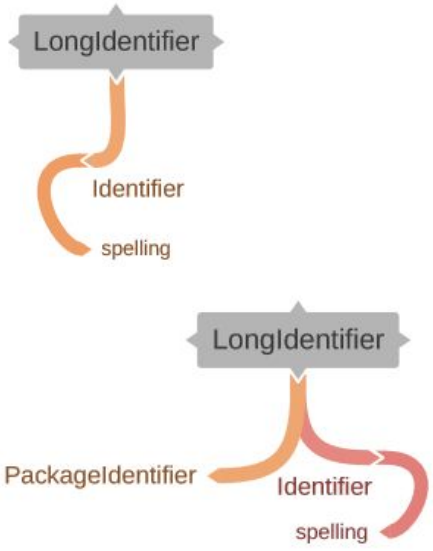


Representaciones de la Categoría Literals, Identifiers y Operators









Extensión realizada a los métodos que permiten visualizar los árboles sintácticos abstractos

Para los métodos que permiten visualizar los AST se extendieron para poder abarcar las nuevas clases creadas para el Triángulo extendido. Para cada una de las clases concretas nuevas se creó un visitor respectivo para que la visualización al AST fuera posible. A continuación se tiene el listado de los nuevos métodos implementados:

- visitCallLoopCases
- visitProcProcFunc
- visitFuncProcFunc
- visitProcFuncs
- visitLoopCasesWhile
- visitLoopCasesUntil
- visitLoopCasesDo
- visitLoopCasesFOR
- visitDoLoopUntil
- visitDoLoopWhile
- visitForLoopDo
- visitForLoopUntil
- visitForLoopWhile
- visitCaseLiterals
- visitCaseRangeCase
- visitSequentialCaseRange
- visitCaseLiteralCHAR
- visitCaseLiteralINT
- visitLParenExpression
- visitLCurlyExpression
- visitLBracketExpression
- visitSequentialSingleDeclaration
- visitCompoundDeclarationPrivate

- visitCompoundDeclarationRecursive
- visitCompoundDeclarationSingleDeclaration
- visitVarSingleDeclarationColon
- visitVarSingleDeclarationSingleDeclaration
- visitPackageDeclaration
- visitSequentialPackageDeclaration
- visitRTypeDenoter
- visitMultipleRecordTypeDenoter
- visitSingleRecordTypeDenoter
- visitLongIdentifier
- visitCompoundIdentifier
- visitDotVname
- visitSubscriptVname

Extensión realizada a los métodos que permiten representar los árboles sintácticos abstractos como texto en XML

Con el fin de representar los árboles sintácticos abstractos como texto en XML, se utilizaron los recursos facilitados por el profesor, los cuales son las clases: Writer y WriterVisitor. Estos recursos originalmente escribían texto en HTML, por lo que se modificó el string que se le pasa por parámetro al método writeLineHTML(), de modo que fuera el formato de XML. Además, se cambió el nombre del método a writeLineXML().

Luego, con el fin de que se lograra realizar un recorrido completo entre los diferentes AST creados en el Parser, fue necesario agregarle métodos al WriterVisitor. Cada uno de los métodos del WriterVisitor, se encarga de recorrer los AST creados, de manera que cada vez que visita los mismos, se encarga de escribir el respectivo texto de XML indicando el production o terminal.

En el caso de los comandos, se agregaron los siguientes métodos de visit:

- visitChooseCommand(ChooseCommand ast, Object o)
- visitCallLoopCases(CallLoopCases ast, Object o)

De los mismos comandos se removió el siguiente método:

- visitWhileCommand()

Del mismo modo, se agregaron los métodos necesarios para recorrer los AST de los diferentes procs de la gramática:

- visitProcProcFunc(ProcProcFunc ast, Object o)
- visitFuncProcFunc(FuncProcFunc ast, Object o)
- visitProcFuncs(ProcFuncs ast, Object o)

Respecto a los loops, se agregaron los siguientes métodos para recorrer sus respectivos AST:

- visitLoopCasesWhile(LoopCasesWhile ast, Object o)
- vistLoopCasesUntil(LoopCasesUntil ast, Object o)
- visitLoopCasesDo(LoopCasesDo ast, Object o)
- visitLoopCasesFOR(LoopCasesFOR ast, Object o)
- visitDoLoopWhile(DoLoopWhile ast, Object o)

Para los cases, se tuvieron que agregar los siguientes métodos para recorrer sus respectivos AST:

- vistCases(Cases ast, Object o)
- visitElseCase(ElseCase ast, Object o)
- visitSequentialCase(SequentialCase ast, Object o)
- vistCaseWhen(CaseWhen ast, Object o)
- vistCaseLiterals(CaseLiterals ast, Object o)
- vistCaseRange(CaseRange ast, Object o)
- visitSequentialCaseRange(SequentialCaseRange ast, Object o)
- vistCaseLiteralCHAR(CaseLiteralCHAR ast, Object o)
- vistCaseLiteralINT(CaseLiteralINT ast, Object o)

Luego, para las expresiones fue necesario agregar los siguientes métodos para recorrer sus AST y escribir el respectivo texto XML:

- visitAssignExpression(AssignExpression ast, Object o)
- visitSecExpression(SecExpression ast, Object o)
- visitOperatorExpression(OperatorExpression ast, Object o)
- visitLParenExpression(LParenExpression ast, Object o)
- vistLCurlyExpression(LCurlyExpression ast, Object o)
- visitLBracketExpression(LBracketExpression ast, Object o)

Asimismo fue necesario agregar métodos para los declarations, con el fin de recorrer sus AST, estos fueron:

- visitPackageDeclaration(PackageDeclaration ast, Object o)
- visitSequentialSingleDeclaration(SequentialSingleDeclaration ast, Object o)
- visitSequentialPackageDeclaration(SequentialPackageDeclaration ast, Object o)
- visitCompoundDeclarationRecursive(CompoundDeclarationRecursive ast, Object o)
- visitCompoundDeclarationPrivate(CompoundDeclarationPrivate ast, Object o)
- visitCompoundDeclarationSingleDeclaration(CompoundDeclarationSingleDeclaration ast, Object o)
- visitVarSingleDeclarationColon(VarSingleDeclarationColon ast, Object o)
- visitVarSingleDeclarationSingleDeclaration(VarSingleDeclarationSingleDeclaration)
- visitVarDeclaration(VarDeclaration ast, Object o)

Los métodos agregados para recorrer los diferentes AST de los type denoters fueron:

- visitMultipleRecordTypeDenoter(MultipleRecordTypeDenoter ast, Object o)
- visitSingleRecordTypeDenoter(SingleRecordTypeDenoter ast, Object o)
- visitTypeDenoterLongIdentifier(TypeDenoterLongIdentifier ast, Object o)
- visitRTypeDenoter(RTypeDenoter ast, Object o)

Se añadieron los siguientes métodos para desplegar los AST de literals, identifiers y operators, igualmente por cada uno se escribe el respectivo texto de XML:

- visitLongIdentifier(LongIdentifier ast, Object o)
- visitCompoundIdentifier(CompoundIdentifier ast, Object o)

Plan de pruebas para validar el compilador

Para poder probar el analizador sintáctico y su correctitud, se utilizó la gramática para generar al menos un caso de cada tipo de producción, de manera que, se escribió un programa en triángulo para cada uno. Esto con el fin de asegurarnos de que el código implementado fuese capaz de procesar y generar un árbol de sintaxis abstracta correcto. Se listaran todos las pruebas, sus resultados esperado y su resultado específico. Todas tienen el mismo objetivo: determinar si se generó el árbol de sintaxis esperado. Algunas pruebas por definición requieren la prueba de sus subpartes, lo cual está denotado por tabulación. Las pruebas positivas son aquellas que se espera se ejecuten sin más error, mientras que las negativas son aquellas que se espera que fallen.

Además de las pruebas planteadas por el equipo se probaron las pruebas aportadas por el profesor, tanto las negativas como las positivas cumplieron con el comportamiento y los resultados esperados. Se utilizaron algunas de esas pruebas aportadas por el profesor para la sección de pruebas negativas sintácticas y léxicas.

Pruebas Positivas Sintácticas

Secondary Expression

Objetivo: determinar si se reconocen los casos de expresiones secundarias y si se genera su respectivo árbol de sintaxis abstracta.
Diseño: if [1=34, ' a ' = ' b '] * {angelo ~ 1} then pass else pass end
Resultados esperados: AST que contenga una expresión secundaria con las respectivas partes.
Resultados observados: Se obtuvó el AST correcto.

Primary Expression

Objetivo: determinar si se reconocen los casos de expresiones primarias y si se genera su respectivo árbol de sintaxis abstracta.

Casos y sus respectivos diseños:
1. Integer-Literal
if 1 then pass else pass end

2. Character-Literal
if ‘a’ then pass else pass end

3. V-name
if angelo[1][0].Atributo then pass else pass end

4. Long-Identifier **(** Actual-Parameter-Sequence **)**
if paquete $ identificador(angelo,jose) then pass else pass end

5. Operator primary-Expression

if * 1 - 2 + 3 then pass else pass end

6. **(** Expression **)**
if (1=2*3) then pass else pass end
7. **{** Record-Aggregate **}**
    1. if {angelo ~ 1*2, ignacio~ 3=4-1} then pass else pass end
    2. if {angelo ~ 1*2} then pass else pass end

8. **[** Array-Aggregate **]**
if [1=34, 'a' = 'b'] then pass else pass end

Resultados esperados: AST que contenga una expresión primaria con las respectivas partes.
Resultados observados: Se obtuvó el AST correcto para cada caso.

Record Aggregate

Objetivo: determinar si se reconocen los casos de Record Aggregate y si se genera su respectivo árbol de sintaxis abstracta.

Casos y sus respectivos dsieños:

1. if {angelo ~ 1*2, ignacio~ 3=4-1} then pass else pass end
2. if {angelo ~ 1*2} then pass else pass end

Resultados esperados: AST que contenga un Record Aggregate, de 1 y 2 componentes para comprobar multiplicidad
Resultados observados: Se obtuvó el AST correcto para cada caso.

Array Aggregate

Objetivo: determinar si se reconocen los casos de Array Aggregate y si se genera su respectivo árbol de sintaxis abstracta.

Casos y sus respectivos diseños:

1. if [1=34, 'a' = 'b'] then pass else pass end
2. if [1=34] then pass else pass end

Resultados esperados: AST que contenga un Array Aggregate, de 1 y 2 componentes para comprobar multiplicidad
Resultados observados: Se obtuvó el AST correcto para cada caso.

VName

Objetivo: determinar si se reconocen los casos de Vname, probando también VarName y si se genera su respectivo árbol de sintaxis abstracta.

Casos y sus respectivos dieños:

1. if pckgAngelo $ angelo then pass else pass end

2. if angelo then pass else pass end

Resultados esperados: AST que contenga un Vname, tanto con como sin identificador de paquete.
Resultados observados: Se obtuvó el AST correcto para cada caso.

Varname
Objetivo: Probar la subproducción de VName, Varname y determinar si se genera su respectivo árbol de sintaxis abstracta.

Casos y sus respectivos diseños:
1. if angelo.ramirez then pass else pass end
2. if angelo.ramirez.ortega then pass else pass end
3. if angelo[1] then pass else pass end
4. if angelo[1][0] then pass else pass end
5. if angelo[1][0].Atributo then pass else pass end

Resultados esperados: AST que contenga un Vname con los seleccionadores en orden lógico.
Resultados observados: Se obtuvó el AST correcto para cada caso.

Declaration

Objetivo: determinar si se reconocen los casos de declaraciones y si se genera su respectivo árbol de sintaxis abstracta.
Diseño: package paquete ~
```
recursive proc angelo() ~ pass end
| func ignacio():Integer ~ 1+2
end;
recursive proc manuel() ~ pass end
| func juan():Integer ~ 1+2
end
end;
pass
```
Resultados esperados: AST que contenga una declaración con las respectivas partes.
Resultados observados: Se obtuvó el AST correcto.

Compound-Declaration

Objetivo: determinar si se reconocen los casos de declaraciones compuestas y si se genera su respectivo árbol de sintaxis abstracta.

Casos y sus respectivos diseños:

Sin paquetes

1.single-Declaration

let var angelo : juan $ Integer in pass end

2.recursive Proc-Funcs end

let
recursive proc angelo() ~ pass end
| proc ignacio() ~ pass end
end
in pass end

3. private Declaration in Declaration end

let private var angelo : Integer in var ignacio : Trejos end in pass end

4.par single-Declaration (| single-Declaration) end

let par var angelo : Integer | var ignacio : Trejos end in pass end

Con Paquetes
1.single-Declaration
package paquete ~ var angelo : Integer end; pass

2. recursive Proc-Funcs end

package paquete ~
recursive proc angelo() ~ pass end
| proc ignacio() ~ pass end
end
end;
pass

3 private Declaration in Declaration end

package paquete ~ private var angelo : Integer in var ignacio : Trejos end end; pass

4. par single-Declaration (| single-Declaration)$^{+}$ end

package paquete ~ par var angelo : Integer | var ignacio : Trejos end end; pass

Resultados esperados: AST que contenga una declaración compuesta con las respectivas partes.
Resultados observados: Se obtuvó el AST correcto.

Proc Func

Objetivo: determinar si se reconocen los casos de procedimientos y funciones, probando proc funcs, el cual contiene ambas en declaraciones secuenciales, tanto usando paquetes como sin usarlos.

Proc Funcs

Casos y sus respectivos diseños:

Sin Paquetes

1.

```
let
recursive proc angelo() ~ pass end
| func ignacio():Integer ~ 1+2
end
in pass end
```

2.

```
let
recursive proc angelo() ~ pass end
| proc ignacio() ~ pass end
end
in pass end
```

Con Paquetes

1.

```
package paquete  ~
recursive proc angelo() ~ pass end
| func ignacio():Integer ~ 1+2
end
end;
pass
```

2.

```
package paquete  ~
recursive proc angelo() ~ pass end
| proc ignacio() ~ pass end
end
end;
pass
```

Resultados esperados: AST que contenga una declaración compuesta procedimientos y/o funciones con y sin paquetes.

Resultados observados: Se obtuvó el AST correcto.

Var Single Declaration

Objetivo: determinar si se reconocen los casos de Declaraciones de Variables y si se genera su respectivo árbol de sintaxis abstracta. Tanto con paquetes como sin.

Casos y sus respectivos diseños:

sin paquete

1. let var angelo : Integer in pass end
2. let var angelo ::= 123 in pass end

con paquetes

1. package paquete ~ var angelo : Integer end; pass
2. package paquete ~ var angelo ::= 123 end; pass

Resultados esperados: AST que contenga un Var Single Declaration.
Resultados observados: Se obtuvó el AST correcto para cada caso.

Actual Parameter Sequence

Objetivo: determinar si se reconocen los casos de Actual Parameter Sequence, probando Proper Actual Parameter Sequence y Actual Parameter, transitoriamente. Esperando que se genere su respectivo árbol de sintaxis abstracta.

Casos y sus respectivos diseños:

Proper Actual Parameter Sequence

if funcion() then pass else pass end

Actual Parameter

if funcion(1, var angelo, proc unProc, func otraFunc) then pass else pass end

Resultados esperados: AST que contenga Actual Parameter Sequence, con un Proper Actual Parameter Sequence compuesto de Actual Parameters
Resultados observados: Se obtuvó el AST correcto para cada caso.

Single Command

Objetivo: determinar si los posibles casos de single command, son reconocidos y correctamente parseado y que sus respectivos AST se creen de acuerdo a lo esperado.
Casos y sus respectivos diseños de los casos de prueba:

V-name **:=** Expression

jose := 1

Long-Identifier **(** Actual-Parameter-Sequence **)**

jose (var jose , proc iden , func as)

pass

let const i ~ 1=1 in pass end

loop Loop-Cases

Detallado en Loop-Cases

let Declaration **in** Command **end**

let const i ~ 1=1 in pass end

if Expression **then** Command **else** Command **end**

if 1=2 then pepe() else pepe() end

choose Expression **from** Cases **end**

Detallado en Cases

Resultados esperados: AST que contenga a los single command con su respectiva estructura, tomando en cuenta cada elemento que lo forma
Resultados observados: Los AST se crean correctamente, tanto la estructura como el orden de las mismas.

Loop-Cases

Objetivo: determinar si los casos de los loop, todos crean sus AST correctamente, con su estructura y orden correspondiente
Casos y sus respectivos diseños de los casos de prueba:

loop while Expression **do** Command **end**

loop while 1=2 do pepe() end

loop until Expression **do** Command **end**

loop until 1=2 do pep() end

loop do Command Do-loop

Detallado en Do-loop

loop for Identifier **from** Expression **to** Expression For-loop

Detallado en For-loop

Resultados esperados: AST que contenga a los casos del loop con su respectiva estructura, tomando en cuenta cada elemento que lo forma. En este caso solo el del while y el del until, dado que los otros son probados en su respectiva producción.
Resultados observados: Los AST se crean correctamente, tanto la estructura como el orden de las mismas.

Do-loop

Objetivo: determinar si todos los casos planteados para la producción de Do-loop en su estructura completa funcionan y crean los AST con la estructura completa
Casos y sus respectivos diseños de los casos de prueba:

loop do Command **while** Expression **end**

loop do pepe() while 1=2 end

loop do Command **until** Expression **end**

loop do pepe() until 1=2 end

Resultados esperados: los AST de todos los casos planteados para la producción de Do-loop en su estructura completa
Resultados observados: Los AST obtenidos cumplen con los resultados esperados, los AST y su representación son creadas correctamente.

For-loop

Objetivo: determinar si para los casos planteados todos los AST que se forman tengan la estructura correcta y su representación sea la esperada
Casos y sus respectivos diseños de los casos de prueba:

loop for Identifier **from** Expression **to** Expression **do** Command **end**

loop for i from 1 to 2 do pepe() end

loop for Identifier **from** Expression **to** Expression **while** Expression **do** Command **end**

loop for i from 1 to 2 while 1=2 do pepe() end

loop for Identifier **from** Expression **to** Expression **until** Expression **do** Command **end**

loop for i from 1 to 2 until 1=2 do pepe() end

Resultados esperados: AST creados correctamente al igual que su representación con la respectiva estructura esperada.
Resultados observados: los AST fueron creados correctamente para cada uno de los casos consultados y la representación era la esperada.

Cases, Case, ElseCase Case-Literals, Case-Range y CaseLiteral

Objetivo: Verificar que todos estos funcionen de forma conjunta ya que en su mayoría de casos se van a usar de la forma que definen los casos de pruebas solo se va a probar para los caso con el objetivo de demostrar que funcionan para los demás.
Casos y sus respectivos diseños de los casos de prueba:

Case

Con Case-literal un Integer-Literal

```
choose a  from  when 12 then pepe() end
```

Con Case-literal un Character-Literal

```
choose a  from  when 'a' then pepe() end
```

Case Case ElseCase

```
choose a from when 'a' then pepe() when 'b' then pepb() else otra() end
```

Case ElseCase

```
choose a  from  when 'a' then pepe()  else pepe() end
```

Case-Literals con varios Case-Range

```
choose a  from  when 1 | 2 | 3 then pepe() when 'a' then pepe()  else
pepe() end
```

Case-Range con dos Case Literal

```
choose a  from  when 1 .. 2 | 2 | 3 then pepe() when 'a' then pepe()
else pepe() end
```

Resultados esperados: Se esperaba obtener los AST correspondientes a todos los casos presentados anteriormente de forma que se pudiera demostrar que con los casos presentados se pueden asegurar que se va a obtener de forma correcta todos los AST que la gramática puede generar para estos casos.
Resultados observados: Los AST generados y sus representaciones fueron las esperadas, de forma que se logra asegurar la creación de las demás posibilidades.

Expression

Objetivo: mostrar que los casos posibles de los AST de Expression son generados correctamente al igual que su representación
Casos y sus respectivos diseños de los casos de prueba:

secondary-Expression

Detallado en la sección de Secondary-Expression

let Declaration **in** Expression

jose := let proc hace (i : as) ~ pass in 1

if Expression **then** Expression **else** Expression

jose := if (1=1+3) then puteol() else puteol()

Resultados esperados: AST creados correctamente, de forma que se logre diferencias a este if y let de los otros y la representación sea correcta
Resultados observados: AST diferentes a los otros let e if y creados correctamente junto con su representación

Single-Declaration

Objetivo: demostrar que los AST y su representación para single-Declaration son creados correctamente.
Casos y sus respectivos diseños de los casos de prueba:

const Identifier **~** Expression

let const i ~ 1=1; const i ~ 1=1 in pass end

var Identifier Var-Single-Declaration

let var i ::= 1=1 in pass end

proc Identifier **(** Formal-Parameter-Sequence **) ~** Command **end**

let proc hace (i : as) ~ pass in pass end

func Identifier **(** Formal-Parameter-Sequence **) :** Type-denoter **~** Expression

let func jose ( ) : is ~ 1-1 in pass end

type Identifier **~** Type-denoter

let type jo ~ se in pass end

Resultados esperados: AST creados correctamente, definiendo en su representación a cada una de sus estructuras que los componen.
Resultados observados: AST fueron creados correctamente, se logra diferenciar bien la representación de una con respecto a otra.

Formal-Parameter-Sequence, proper-Formal-Parameter-Sequence y Formal-Parameter

Objetivo: Mostrar que los AST junto con su representación se creen de forma correcta y por medio de algunos ejemplos mostrar que si funcionan con estas condiciones de parámetros, para el resto de posibles casos también van a funcionar.
Casos y sus respectivos diseños de los casos de prueba:

Sin proper-Formal-Parameter-Sequence

let func jose ( ) : is ~ 1-1 in pass end

Con proper-Formal-Parameter-Sequence con Identifier : Type-denoter de Formal-Parameter

let func jose (jo : o): is ~ 1-1 in pass end

Varios Formal Parameter con **func** Identifier **(** Formal-Parameter-Sequence **)** y Identifier **:** Type-denoter de Formal-Parameter

let func jose (func is ( ) : ty , jose : o): is ~ 1 in pass end

Con **var** Identifier **:** Type-denoter y **proc** Identifier **(** Formal-Parameter-Sequence **)** de Formal-Parameter

let func jose (var jose : cs , proc hace ( )): is ~ 1 in pass end

Resultados esperados: AST con los que se puedan demostrar que con los casos propuestos se puede asegurar que el resto de posibilidades funcionan y que la representación sea la adecuada
Resultados observados: AST con la estructura y representación esperada.

Type-denoter

Objetivo: Verificar que los AST junto con su representación son creados correctamente para todos los casos

Casos y sus respectivos diseños de los casos de prueba:
Long-Identifier

let func jose (jo : o): is ~ 1-1 in pass end

array Integer-Literal **of** Type-denoter

let type jo ~ array 123 of jo in pass end

record Record-Type-denoter **end**

let type jo ~ record jo : se end in pass end

Varios Record-Type-denoter

let type jo ~ record jo : se , se : pa end in pass end

Resultados esperados: AST y representaciones correctas de acuerdo a los casos propuestos
Resultados observados: AST fueron creados y representados de la forma esperada, y se logra demostrar que con los casos propuestos se cumplen para el resto de posibilidades

Pruebas Negativas Sintácticas

Como ejemplo pruebas negativas se tienen los siguientes casos:
loop
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos de loop
Diseño:pere

loop for 2 from 1 to 2 do pass end

loop n from 1 to 2 do pass end

loop for n from 1 2 do pass end

loop for n from 1 to 2 do pass

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que hace falta el end.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico, como por ejemplo que falta el end en la última o que falta el to en la penúltima.

choose

Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos choose y se despliegue el mensaje correcto correspondiente

Diseño:

```
choose 1 from
  when then putint (1
en
! Error: falta el literal
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que hace falta el literal.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico, en este caso la falta de un literal.

if

Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos if y se despliegue el mensaje correcto correspondiente

Diseño:

```
if (1=1) puteol() else puteol() end

! Error: Faltó el then (puso paréntesis, como en C)
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que se esperaba un then.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico, en este caso la falta de un then.

Let

Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos let y se despliegue el mensaje correcto correspondiente

Diseño:

```
let var i : Char;
var j : Integer;
const k ::= 8;
type dia ~ Integer

put(i);
put(i)
```

end

! Error: Falta el in

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Loop Do Until
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos loop do until y se despliegue el mensaje correcto correspondiente

Diseño:
do pass until 4=4 end

! Error: Falta el loop

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Loop Do While
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos loop do while y se despliegue el mensaje correcto correspondiente

Diseño:
loop do do pass;pass while 3=3 end

! Error: Hay 2 do

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Loop For

Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos loop for y se despliegue el mensaje correcto correspondiente

Diseño:

loop i from 5 to 6 do pass;pass end

! falta for

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Loop For Until
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos loop until for y se despliegue el mensaje correcto correspondiente

Diseño:

loop for i from 5 to 6 i < 5 do pass;pass end

!falta until o while

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Loop For Until
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos loop for until y se despliegue el mensaje correcto correspondiente

Diseño:

loop for i from 5 to 6 i < 5 do pass;pass end

!falta until o while

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Loop Until Do
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos loop until do y se despliegue el mensaje correcto correspondiente

Diseño:

```
until 2=2 do pass;pass end

! Error: Falta el loop
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Loop While Do
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos loop while do y se despliegue el mensaje correcto correspondiente

Diseño:

```
while 1=1 do pass; pass end

! Error vieja sintaxis de while
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Package
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos package y se despliegue el mensaje correcto correspondiente

Diseño:

```
myP ~
        var a: Integer
end;
let
        var b: Integer;
        b:= myP $ a ! esto no tiene la forma de una declaración
```

```
in
        pass
end

!Error: falta el package
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Package Invocation
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos que contienen identificadores de paquetes y se despliegue el mensaje correcto correspondiente

Diseño:

```
package myP ~
        var a: Integer
end;
let
        var b: Integer
in
        b:= myP  a;
        putint(b)

end

!Error: falta el $
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Par
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos que contienen comandos par y se despliegue el mensaje correcto correspondiente

Diseño:

```
let
```

```
        var i: Integer | var k: Integer end
in
        putint(i);
        put(i)
end

! Error: falta par
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Pass
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos que contienen comandos pass y se despliegue el mensaje correcto correspondiente. Probando adicionalmente la eliminación de otros comandos vacíos.

Diseño:

```
        puteol();

        ! Error: Después del ";" faltaría un pass
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Private
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos que contienen private y se despliegue el mensaje correcto correspondiente.

Diseño:

```
        let
                private
                        var a::=5;
                        var b::=15
                ! in
                        var c::=a*b*2
                end ;
                var d::= 5
        in
```

```
        putint(c)
    end

    ! Error: Falta el in del private
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Recursive
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos que contienen recursive y se despliegue el mensaje correcto correspondiente.

Diseño:

```
    let
        recursive
            proc doble ( var i : Integer ) ~
                i := i*2
            end
        end
    in
        pass
    end

    ! Error: Solo hay una alternativa, deben haber dos
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Var
Objetivo: probar que el programa no va a crear la representación de los AST cuando hay errores en los comandos que contienen var y se despliegue el mensaje correcto correspondiente.

Diseño:

```
    var j::=5
```

! Error: No se puede declarar fuera de un let o un local

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Pruebas Positivas Léxicas

Par
Objetivo: probar que el analizador léxico reconozca a par como una palabra reservada.

Diseño:

```
let
        par var i: Integer | var k: Integer | var t: Char end
in
        putint(i);
        put(t)
end
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

Pass
Objetivo: probar que el analizador léxico reconozca a pass como una palabra reservada.

Diseño:

```
pass
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

Private

Objetivo: probar que el analizador léxico reconozca a private como una palabra reservada.

Diseño:

```
let
```

```
        private
            var a::=5;
            var b::=15
        in
            var c::=a*b*2
        end;
        var d::= 5
in
        putint(c)
end
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

Recursive
Objetivo: probar que el analizador léxico reconozca a recursive como una palabra reservada.

Diseño:

```
    let
        recursive
            func doble(var i : Integer) : Integer ~
                triplicar(i)/3*2
            |
            proc triplicar(var i : Integer) ~
                i := i + doble(i)
            end
            |
            proc duplicar(var i : Integer) ~
                i := doble(i)
            end
        end
    in
        pass
    end
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

To

Objetivo: probar que el analizador léxico reconozca a to como una palabra reservada.

Diseño:

```
loop for i from 5 to 6 until i < 5 do pass;pass end
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

Until
Objetivo: probar que el analizador until reconozca a par como una palabra reservada.

Diseño:

```
loop for i from 5 to 6 until i < 5 do pass;pass end

!Error en casing del until
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

When
Objetivo: probar que el analizador léxico reconozca a when como una palabra reservada.

Diseño:

```
choose 1 from

  when 0 .. 5 | 8 then putint (1)

  when 7 then putint(2)

end
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

|
Objetivo: probar que el analizador léxico reconozca a | como un token.

Diseño:

```
choose 1 from
```

```
  when 0 .. 5 | 8 then putint (1)

  when 7 then putint(2)

end

! Error | no existe
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

::=
Objetivo: probar que el analizador léxico reconozca a ::= como un token.

Diseño:

```
let
        var j ::= 5;
in
        pass
end
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

$
Objetivo: probar que el analizador léxico reconozca a $ como un token.

Diseño:

```
package myP ~
        var a: Integer
end;
let
        var b: Integer
in
        b:= myP $ a;
        putint(b)

end
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

..
Objetivo: probar que el analizador léxico reconozca a .. como un token.

Diseño:

```
choose 1 from

  when 0 .. 5 | 8 then putint (1)

  when 7 then putint(2)

end
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

Begin
Objetivo: probar que el analizador léxico reconozca a begin como un identificador común.

Diseño:

```
let
  var begin: Integer
in
  putint (begin);
  puteol ()
end

! OK: begin ya no es palabra reservada
! y el comando puede ser compuesto
```

Resultados esperados: AST y representaciones correctas de acuerdo a lo esperado del analizador sintáctico.
Resultados observados: AST fueron creados y representados de la forma esperada.

Pruebas Negativas Léxicas

Par
Objetivo: probar que el programa no trate a cosas cercanas a par como Par o PAR como un token. Respetando la diferencia en casing.

Diseño:

```
let
        Par var i: Integer | var k: Integer | var t: Char end
in
        putint(i);
```

```
        put(t)
    end

    ! Error en casing
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Pass

Objetivo: probar que el programa no trate a cosas cercanas a pass como Pass o PASS como un token. Respetando la diferencia en casing.

Diseño:

```
    Pass
    ! Error en casing
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Private

Objetivo: probar que el programa no trate a cosas cercanas a private como Private o PRIBATE como un token. Respetando la diferencia en casing.

Diseño:

```
    let
        PRIBATE
            var a::=5;
            var b::=15
        in
            var c::=a*b*2
        end;
        var d::= 5
    in
        putint(c)
    end

    ! Casing
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Recursive
Objetivo: probar que el programa no trate a cosas cercanas a recursive como Recursive o RECURSIVE como un token. Respetando la diferencia en casing.

Diseño:

```
let
    RECURSIVE
        func doble(var i : Integer) : Integer ~
            triplicar(i)/3*2
        |
        proc triplicar(var i : Integer) ~
            i := i + doble(i)
        end
        |
        proc duplicar(var i : Integer) ~
            i := doble(i)
        end
    end
in
    pass
end

! Casing
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

To
Objetivo: probar que el programa no trate a cosas cercanas a to como To o TO como un token. Respetando la diferencia en casing.

Diseño:

```
loop for i from 5 To 6 until i < 5 do pass;pass end

!Casing del to
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Until
Objetivo: probar que el programa no trate a cosas cercanas a until como Until o Until como un token. Respetando la diferencia en casing.

Diseño:

```
loop for i from 5 to 6 Until i < 5 do pass;pass end

!Error en casing del until
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

When
Objetivo: probar que el programa no trate a cosas cercanas a when como When o WHEN como un token. Respetando la diferencia en casing.

Diseño:

```
choose 1 from

  When 0 .. 5 | 8 then putint (1)

  WHEN 7 then putint(2)

end

! Error de casing de when
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Objetivo: probar que el programa no trate a cosas cercanas a | como ||. Respetando la diferencia en casing.

Diseño:

```
choose 1 from

  when 0 .. 5 || 8 then putint (1)

  when 7 then putint(2)

end

! Error || no existe
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

::=

Objetivo: probar que el programa no trate a cosas cercanas a ::= como ::== como un token. Respetando la diferencia en escritura.

Diseño:

```
let
        var j ::== 5;
in
        pass
end

! Error ::== no existe
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

$

Objetivo: probar que el programa no trate a cosas cercanas a $ como $$ como un token. Respetando la diferencia en escritura.

Diseño:

```
package myP ~
        var a: Integer
end;
let
        var b: Integer
in
        b:= myP $$ a;
        putint(b)

end

!Error $$ no existe
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

..
Objetivo: probar que el programa no trate a cosas cercanas a .. como ... como un token. Respetando la diferencia en escritura.

Diseño:

```
choose 1 from

  when 0 ... 5 | 8 then putint (1)

  when 7 then putint(2)

end

! … no existe
```

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Begin
Objetivo: probar que el programa no trate a begin como una palabra reservada, dado que ya no lo es.

Diseño:

begin puteol() end

! Error: Begin no existe

Resultados esperados: Que en la consola nos indique el error que hubo y la línea y por ende que los AST no se creen. Con casos como el último se espera que la consola indique que no se esperaba lo encontrado.
Resultados observados: Consola indica el error que hubo y la línea, AST no se crean y los casos que lo ameriten indican el error específico.

Análisis de la ‘cobertura’ del plan de pruebas

El plan de pruebas planteado para este proyecto cubre todos los aspectos que puede generar la gramática. Es decir que cualquier combinación de código que sea generable por la gramática es probada en este plan de pruebas. De esta manera se asegura que el compilador pueda reconocer cualquier programa escrito en Triángulo extendido y que lo pueda entender y manejar de manera correcta, esto incluye la generación del XML, HTML y árboles de sintaxis abstractos.

Sin embargo, este plan de pruebas no cubre la posibilidad de que el compilador acepte un programa que no es generable mediante la gramática. Es decir no se valida que el compilador únicamente comprenda lo generado por la gramática ya que esto resulta en un plan de pruebas de dimensiones mucho más grandes, lo cual no es relevante para esta parte del desarrollo del compilador. La cobertura diseñada para este plan está pensada para cubrir las áreas que vayan de acorde al propósito de esta fase del proyecto para asegurar el completo éxito del compilador.

Discusión y análisis de los resultados obtenidos.

El desarrollo de este proyecto nos permitió entender muchas cosas. Una de ellas es entender el esfuerzo que implica el tener que modificar un código escrito por alguien más para agregar más funcionalidades ya que esto conlleva el completo entendimiento del código que puede ser algo muy tedioso y el análisis de los paquetes para entender dónde deben ser hechas las modificaciones para que funcione de manera correcta y limitar la generación de nuevas pulgas al mínimo.

En cuanto a resultados obtenidos, se tiene satisfacción completa ya que el compilador trabaja correctamente y cumple con todos los requerimientos planteados para esta entrega, no solo trabaja correctamente, sino que los cambios hechos en la gramática trabajan de manera muy eficiente en el compilador por lo que no se da un mal uso o gasto de recursos computacionales.

Finalmente, el desarrollar un compilador a base de uno ya creado permite ver cómo se debe programar un compilador de manera correcta ya que se tiene poca experiencia en esto y permite realizar las modificaciones bajo el mismo marco de trabajo y con el mismo

nivel de calidad que el compilador base, a diferencia de realizar un compilador desde cero, que si bien permite mucho más aprendizaje del código de un compilador también se tiende a tener muchos errores y ser poco robusto.

Reflexión

Es complicado modificar código que uno no ha escrito, personalmente es nuestra primera experiencia con este tipo de dificultad. Se tiene que dedicar una gran cantidad de tiempo a comprender el código y cuales son las convenciones que usa el código escrito, aunque en este caso no sucedió es muy común encontrar código sucio o difícil de entender y ademas que no usan buenas practicas de programacion.

Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo.

Jose Pablo Alvarado Flores

- Factorización de la gramática
- Cambios a Token
- Cambios a Scanner
- Implementación de la gramática en el Parser
- Plan de pruebas
- Documentación

Ricardo Bonilla Morales

- Factorización de la gramática
- Cambios a Token
- Cambios a Scanner
- Implementación de la gramática en el Parser
- XML
- Documentación

Guilliano D’ Ambrosio Sosa

- Factorización de la gramática
- Cambios a Token
- Cambios a Scanner
- HTML
- Documentación

Angelo Ortega Ramirez

- Factorización de la gramática
- Cambios a Token

- Cambios a Scanner
- Implementación de la gramática en el Parser
- Plan de pruebas
- Documentación

Cómo se compila el programa

Para compilar un programa se tienen dos opciones. La primera es una vez se encuentra en el IDE se puede cargar un archivo desde la pestaña FILE al presionar la opción LOAD, debe tenerse en cuenta que el archivo debe ser .TRI, luego de esto se puede compilar con F5 o abriendo la pestaña TRIANGLE y presionando COMPILE. La segunda manera es ingresar a la pestaña FILE luego hacer un nuevo archivo con la opción NEW, luego de esto se procede a escribir el código deseado y luego procede con los pasos de la opción anterior para compilar el programa desde la pestaña TRIANGLE

Como se ejecuta el programa

Se le da click al archivo ejecutable “IDE-Triangle.jar” con esto se ejecuta el IDE y ya se puede proceder a compilar los programas o bien, si no se tiene el ejecutable, para ejecutar el programa se recomienda usar NetBeans. Primero se abre el proyecto en NetBeans luego se procede a presionar el botón Run Project que está representado con un triángulo verde. Con esto se ejecuta el IDE de triángulo extendido listo para analizar sintácticamente un código del lenguaje. Este IDE permite abrir un archivo .TRI o bien crear uno en el mismo IDE para luego guardarlo y compilarlo.