Integrantes:

- Daniela Rodríguez C-311
- Carlos Carret C-312
- Belsai Arango C-311

0.1 Parser

El lenguaje de dominio específco DSP. Se modela a partir de un conjunto de símbolos, cuya implementación se encuentra en la clase Symbol:

Parser > Symbol.py > Symbol

```python
class Symbol(object):
    def __init__(self, name, grammar):
        self.name = name
        self.grammar = grammar

    def __repr__(self):
        return repr(self.name)

    def __str__(self):
        return self.name

    def __add__(self, other):
        if isinstance(other, Symbol):
            return Sentence(self, other)
        raise TypeError(other)

    def __or__(self, other):
        if isinstance(other, Sentence):
            return SentenceList(Sentence(self), other)
        raise TypeError(other)

    def __len__(self):
        return 1

    @property
    def is_epsilon(self):
        return False
```

Los símbolos facilitan la formación de oraciones al agruparse con el operador +, permitiendo el reconocimiento de la cadena especial epsilon a través de la propiedad is epsilon. Además, posibilita el acceso a la gramática en la que se definió mediante el campo *grammar* que contiene cada instancia, así como la consulta de su notación a través del campo *name*.

Esta también permite la definición de símbolos terminales y no terminales. En el caso de los no terminales, su modelación se encuentra en la clase NonTerminal, la cual extiende a la clase Symbol.

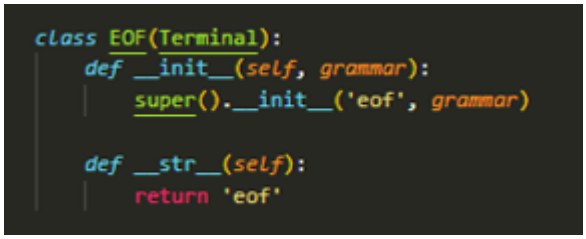
```
class NonTerminal(Symbol):
    def __init__(self, name, grammar):
        super().__init__(name, grammar)
        self.Productions = []

    def __str__(self):
        return self.name

    def __mod__(self, other):
        if isinstance(other, Sentence):
            p = Production(self, other)
            self.grammar.add_production(p)
            return self
        if isinstance(other, tuple):
            if len(other) == 2:
                other += (None,) * len(other[0])
            # Debe definirse una regla por cada símbolo de la producción
            if isinstance(other[0], Symbol) or isinstance(other[0], Sentence):
                p = AttributeProduction(self, other[0], other[1:])
            else:
                raise Exception("")
            self.grammar.add_production(p)
            return self
        if isinstance(other, Symbol):
            p = Production(self, Sentence(other))
            self.grammar.add_production(p)
            return self
        if isinstance(other, SentenceList):
            for s in other:
                p = Production(self, s)
                self.grammar.add_production(p)
            return self
        raise TypeError(other)

    @property
```

```
    @property
    def is_terminal(self):
        return False

    @property
    def is_non_terminal(self):
        return True

    @property
    def is_epsilon(self):
        return False
```

Para reconocer sus producciones se tiene el campo productions de cada instancia. También se pueden añadir las producciones a través del operador % =. Esta incluye las propiedades *isnonterminal* e *isterminal* que devolverán True o False respectivamente. Esto último se añade de igual manera para los símbolos terminales, cuya modelación se encuentra en la clase Terminal, la cual a igual que NonTerminal extiende a Symbol.

```python
from .Symbol import Symbol

class Terminal(Symbol):
    def __int__(self, name, grammar):
        super().__init__(name, grammar)
        self.Productions = []

    def __str__(self):
        return self.name

    @property
    def is_terminal(self):
        return True

    @property
    def is_non_terminal(self):
        return False

    @property
    def is_epsilon(self):
        return False

class EOF(Terminal):
    def __init__(self, grammar):
        super().__init__('eof', grammar)

    def __str__(self):
        return 'eof'
```

La clase EOF modela el símbolo de fin de cadena, cuyo comportamiento se hereda al extender la clase Terminal.



Las oraciones y formas oracionales del lenguaje se logran a través de la clase Sentence, siendo un conjunto de terminales y no terminales.

```python
class Sentence(object):
    def __init__(self, *args):
        self._symbols = tuple(x for x in args if not x.is_epsilon)
        self.hash = hash(self._symbols)

    def __len__(self):
        return len(self._symbols)

    def __add__(self, other):
        if isinstance(other, Symbol):
            return Sentence(*(self._symbols + (other,)))
        if isinstance(other, Sentence):
            return Sentence(*(self._symbols + other._symbols))

    def __or__(self, other):
        if isinstance(other, Sentence):
            return SentenceList(self, other)
        if isinstance(other, Symbol):
            return SentenceList(self, Sentence(other))

    def __str__(self):
        return ("%s " * len(self._symbols) % tuple(self._symbols)).strip()
    def __iter__(self):
        return iter(self._symbols)
    def __getitem__(self, index):
        return self._symbols[index]
    def __eq__(self, other):
        return self._symbols == other._symbols
    def __hash__(self):
        return self.hash
    @property
    def is_epsilon(self):
        return False
```

Con esta se conoce la longitud de la oración, además se accede a los símbolos que componen la oración a través del campo symbols de cada instancia, y se puede conocer si dicha oración está completamente vacía a través de la propiedad is epsilon. Mediante el operador + se puede obtener la concatenación con un símbolo u otra oración. Para definir las producciones que tengan la misma cabecera en una única sentencia, se emplea el agrupamiento de oraciones usando el operador |, que se maneja con la clase SentenceList.

```
class SentenceList(object):
    def __init__(self, *args):
        self._sentences = list(args)

    def add(self, symbol):
        if not symbol and (symbol is None or not symbol.is_epsilon):
            raise ValueError(symbol)
        self._sentences.append(symbol)

    def __or__(self, other):
        if isinstance(other, Sentence):
            self.add(other)
            return self

        if isinstance(other, Symbol):
            return self | Sentence(other)

    def __iter__(self):
        return iter(self._sentences)
```

En la clase Epsilon se modela tanto la cadena vacía como el símbolo que la representa: ϵ. Dicha clase extiende las clases Terminal y Sentence por lo que adopta el comportamiento de ambas, sobreescribiendo la implementación del método *isepsilon* para indicar que toda instancia de la clase reprensenta a epsilon.

```
class Epsilon(Terminal, Sentence):
    def __init__(self, grammar):
        super().__init__('epsilon', grammar)

    def __hash__(self):
        return hash("")
    def __len__(self):
        return 0
    def __str__(self):
        return "e"
    def __repr__(self):
        return 'epsilon'
    def __iter__(self):
        yield from ()
    def __add__(self, other):
        return other
    def __eq__(self, other):
        return isinstance(other, (Epsilon,))

    @property
    def is_epsilon(self):
        return True
```

La clase Production refleja las producciones. Se accede a la cabecera y cuerpo de cada producción mediante los campos left y right. Para consultar si la producción es de la forma $X \to \epsilon$ se emplea la propiedad *isepsilon* y para bifurcar la producción en cabecera y cuerpo se hace uso de las asignaciones: left, right = production.

```python
class Production(object):
    def __init__(self, non_terminal, sentence):
        self.Left = non_terminal
        self.Right = sentence

    def __str__(self):
        return '%s := %s' % (self.Left, self.Right)

    def __repr__(self):
        return '%s -> %s' % (self.Left, self.Right)

    def __iter__(self):
        yield self.Left
        yield self.Right

    def __eq__(self, other):
        return isinstance(other, Production) and self.Left == other.Left and self.Right == other.Right

    def __hash__(self):
        return hash((self.Left, self.Right))

    @property
    def is_epsilon(self):
        return self.Right.IsEpsilon
```

La modelación de las gramáticas se realiza en la clase Grammar. Sus funcionalidades básicas son definir de la gramática los símbolos terminales, a través de los métodos terminal y terminals y los no terminales mediante non terminal y non terminals, además de denotar las producciones de la gramática a partir de la aplicación del operador % = entre no terminales y oraciones. A su vez, se puede acceder a todas las producciones mediante el campo Productions de cada instancia. Se tienen los terminales y no terminales mediante los campos Terminals y NonTerminals , y al símbolo inicial, epsilon y fin de cadena(EOF) a través de los campos StartSymbol, Epsilon y EOF.

```python
class Grammar:
    def __init__(self):
        self.Productions = []
        self.pType = None
        self.Non_terminals = []
        self.Terminals = []
        self.Start_symbol = None
        self.Epsilon = Epsilon(self)
        self.Eof = EOF(self)
        self.SymbolDict = {'eof': self.Eof}

    def non_terminal(self, name, start_symbol=False):
        if not name:
            raise Exception("Empty")
        term = NonTerminal(name, self)
        if start_symbol:
            if self.Start_symbol is None:
                self.Start_symbol = term
            else:
                raise Exception('Cannot define more than one start symbol')
        self.Non_terminals.append(term)
        self.SymbolDict[name] = term
        return term

    def non_terminals(self, names):
        aux = tuple(self.non_terminal(i) for i in names.strip().split())
        return aux

    def add_production(self, production):
        if len(self.Productions) == 0:
            self.pType = type(production)
        production.Left.Productions.append(production)
        self.Productions.append(production)
```

```python
def terminal(self, name):
    if not name:
        raise Exception('Empty')
    term = Terminal(name, self)
    self.Terminals.append(term)
    self.SymbolDict[name] = term
    return term

def terminals(self, names):
    aux = tuple(self.terminal(i) for i in names.strip().split())
    return aux

def __getitem__(self, item):
    try:
        return self.SymbolDict[item]
    except KeyError:
        return None

def copy(self):
    g = Grammar()
    g.Productions = self.Productions.copy()
    g.Non_terminals = self.Non_terminals.copy()
    g.Terminals = self.Terminals.copy()
    g.pType = self.pType
    g.Start_symbol = self.Start_symbol
    g.Epsilon = self.Epsilon
    g.Eof = self.Eof
    g.SymbolDict = self.SymbolDict.copy()
    return g
```

Para el manejo de la pertenencia o no de epsilon a un conjunto se emplea la clase ContainerSet, la cual funciona como un conjunto de símbolos, posibilitando consultar la pertenencia de epsilon al conjunto. Las operaciones que modifican el conjunto devuelven si hubo cambio o no. Puede actualizarse con la adición de elementos individuales, con el método add, o a partir de otro conjunto, mediante update y hard update.

```python
class ContainerSet:
    def __init__(self, *values, contains_epsilon=False):
        self.set = set(values)
        self.contains_epsilon = contains_epsilon

    def add(self, value):
        n = len(self.set)
        self.set.add(value)
        return n != len(self.set)

    def extend(self, values):
        change = False
        for value in values:
            change |= self.add(value)
        return change

    def set_epsilon(self, value=True):
        last = self.contains_epsilon
        self.contains_epsilon = value
        return last != self.contains_epsilon

    def update(self, other):
        n = len(self.set)
        self.set.update(other.set)
        return n != len(self.set)

    def epsilon_update(self, other):
        return self.set_epsilon(self.contains_epsilon | other.contains_epsilon)

    def hard_update(self, other):
        return self.update(other) | self.epsilon_update(other)
```

Por otra parte, el conjunto First de una forma oracional se define como:

- $First(w) = x \in V_t | w \rightarrow^* x\alpha, \alpha \in (V_t \cup V_n)*$
- $\cup\{\epsilon\}$, si w $\rightarrow^* \epsilon$
- $\cup\{\}$,en otro caso.

Se computa para los símbolos terminales, no terminales y producciones haciendo uso de un método de punto fijo. Para ello los firsts se inicializan vacíos y mediante las siguientes reglas se actualizan con la aplicación de forma incremental:

- Si $X \to W_1|W_2|...|W_n$ entonces por definición: $First(X) = \cup_i First(W_i)$
- Si $X \to \epsilon$ entonces $\epsilon \in First(X)$
- Si W = xZ donde x es un símbolo terminal, entonces $First(W) = \{x\}$
- Si W = Y Z donde Y es un símbolo no terminal y Z una forma oracional, entonces $First(Y) \subseteq First(W)$
- Si W = Y Z y $\epsilon \in First(Y)$ entonces $First(Z) \subseteq First(W)$

El cálculo de los firsts se da por terminado cuando finalice una iteración sin que se produzcan cambios. Para la implementación de dicho algoritmo se tiene el método compute local first, que calcula el First(alpha), siendo alpha una forma oracional.

```
def compute_local_first(firsts, alpha):
    first_alpha = ContainerSet()
    try:
        alpha_is_epsilon = alpha.is_epsilon
    except:
        alpha_is_epsilon = False
    if alpha_is_epsilon:
        first_alpha.set_epsilon()
    else:
        for item in alpha:
            first_symbol = firsts[item]
            first_alpha.update(first_symbol)
            if not first_symbol.contains_epsilon:
                break
        else:
            first_alpha.set_epsilon()
    return first_alpha
```

Con el método compute firsts se calculan todos los conjuntos firsts actualizando a los conjuntos iniciales según los resultados de aplicar compute local first en cada producción

```python
def compute_firsts(g):
    firsts = {}
    change = True

    for terminal in g.Terminals:
        firsts[terminal] = ContainerSet(terminal)

    for non_terminal in g.Non_terminals:
        firsts[non_terminal] = ContainerSet()
    while change:
        change = False

        for production in g.Productions:
            x = production.Left
            alpha = production.Right
            first_x = firsts[x]

            try:
                first_alpha = firsts[alpha]
            except:
                first_alpha = firsts[alpha] = ContainerSet()
            local_first = compute_local_first(firsts, alpha)

            change |= first_alpha.hard_update(local_first)
            change |= first_x.hard_update(local_first)


    return firsts
```

Una gramática atributada es una tupla $<$ G, A, R $>$ donde:

- G $=<$ S, P, N, T $>$ es una gramática libre del contexto,
- A es un conjunto de atributos de la forma $X * \alpha$ donde $X \in N \cup T$ y α es un identificador único entre todos los atributos del mismo símbolo y,
- R es un conjunto de reglas de la forma $< p_i, r_i >$ donde $p_i \in P$ es una producción $X \to Y_1, ..., Y_n$ y r_i es una regla de la forma:
    1. $X * \alpha = f(Y_1 * \alpha_1, ..., Y_n * \alpha_n)$, o
    2. $Y_i * \alpha = f(Y_1 * \alpha_1, ..., Y_n * \alpha_n)$

Los atributos se dividen en dos conjuntos disjuntos atributos heredados y atributos sintetizados, como es el caso de α en (1) y en (2) respectivamente.

Las condiciones suficientes para que una gramática sea evaluable son:

- Una gramática atributada es $s - atributada$ si y solo si, para toda regla r_i asociada a una producción $X \to Y_1, ..., Y_n$, se cumple que r_i es de la forma: $X * \alpha = f(Y_1 \Delta \alpha_1, ..., Y_n * \alpha_n)$.
- Una gramática atributada es $l - atributada$ si y solo si, toda regla r_i asociada a una producción $X \to Y_1, ..., Y_n$, es de una de las siguientes formas:
    1. $X * \alpha = f(Y_1 * \alpha_1, ..., Y_n * \alpha_n)$, o
    2. $Y_i * \alpha = f(Y_1 * \alpha_1, ..., Y_n * \alpha_n)$

Por tanto se añade una nueva clase AttributeProduction:

```python
class AttributeProduction(Production):
    def __init__(self, non_terminal, sentence, attributes):
        if not isinstance(sentence, Sentence) and isinstance(sentence, Symbol):
            sentence = Sentence(sentence)
        super(AttributeProduction, self).__init__(non_terminal, sentence)
        self.attributes = attributes

    def __str__(self):
        return '%s := %s' % (self.Left, self.Right)

    def __repr__(self):
        return '%s -> %s' % (self.Left, self.Right)

    def __iter__(self):
        yield self.Left
        yield self.Right

    @property
    def is_epsilon(self):
        return self.Right.IsEpsilon
```

Con esta clase se tienen las producciones de las gramáticas atributadas. Cada una de estas producciones se componen por un símbolo no terminal como cabecera, accesible a través del campo Left, una oración como cuerpo, a través del campo Right y un conjunto de reglas para evaluar los atributos, accesible a través del campo atributes.

Se implementó la clase Item para modelar los items del parser LR(1), cuyos lookaheads se almacenarán haciendo uso del parámetro lookaheads. Cada item tiene definido una función Preview, la cual devuelve todas las posibles cadenas que resultan de concatenar lo que queda por leer del item tras saltarse x símbolos con los posibles lookaheads, que resultan de calcular el first de estas cadenas.

```python
    @property
    def is_reduce_item(self):
        return len(self.Production.Right) == self.Pos

    @property
    def next_symbol(self):
        if self.Pos < len(self.Production.Right):
            return self.Production.Right[self.Pos]
        else:
            return None

    def next_item(self):
        if self.Pos < len(self.Production.Right):
            return Item(self.Production, self.Pos + 1, self.Lookaheads)
        else:
            return None

#Esta devuelve todas las posibles cadenas que resultan de concatenar
# _"lo que queda por leer del item tras saltarse `skip=1` símbolos"_
# con los posibles lookaheads.
# Esta función nos será útil, pues sabemos que el lookahead de los items LR(1)
# se obtiene de calcular el `first` de estas cadenas
    def preview(self, skip=1):
        return [ self.Production.Right[self.Pos + skip:] + (lookahead,) for lookahead in self.Lookaheads]

    def center(self):
        return Item(self.Production, self.Pos)
```

La clausura se obtiene a partir de la función expand, que recibe un item LR(1) y retorna un conjunto de items que sugiere incorporar debido a la presencia de un . delante de un no terminal.

```
@staticmethod
def compress(items):
    centers = {}
    for item in items:
        center = item.center()
        try:
            lookaheads = centers[center]
        except KeyError:
            centers[center] = lookaheads = set()
        lookaheads.update(item.Lookaheads)
    return {Item(x.Production, x.Pos, set(lookahead)) for x, lookahead in centers.items()}

@staticmethod
def expand(item, firsts):
    next_symbol = item.next_symbol
    if next_symbol is None or not next_symbol.is_non_terminal:
        return []
    lookaheads = ContainerSet()

    for p in item.preview():
        for first in compute_local_first(firsts, p):
            lookaheads.add(first)
    _list = []
    for production in next_symbol.Productions:
        _list.append(Item(production, 0, lookaheads))
    return _list
```

También se tiene la función compress, que recibe un conjunto de items LR(1) y devuelve dicho conjunto, pero combinando los lookaheads de los items con mismo centro.

Teniendo en cuenta ambas funciones, se creó la función de clausura utilizando la técnica de punto fijo, basándose en lo siguiente:

CL(I) = I$\cup\{X \to .\beta, b\}$ tales que:

- $Y \to \alpha.X\beta$, $c \in$ CL(I)

- $b \in First(\beta c)$

```
def goto_lr1(self, items, symbol, firsts=None, just_kernel=False):
    assert just_kernel or firsts is not None, "`firsts` must be provided if `just_kernel=False`"
    items = frozenset(item.next_item() for item in items if item.next_symbol == symbol)
    return items if just_kernel else self.closure_lr1(items, firsts)

def closure_lr1(self, items, firsts):
    closure = ContainerSet(*items)
    changed = True
    while changed:
        new_items = ContainerSet()
        # por cada item hacer expand y añadirlo a new_items
        for item in closure:
            e = self.expand(item, firsts)
            new_items.extend(e)
        changed = closure.update(new_items)
    return self.compress(closure)
```

Por otro lado se tiene la implementación de la función goto, que cumple que:
Goto(I, X) = CL($\{Y \to \alpha X.\beta, c | Y \to \alpha.X\beta, c \in I\}$).

Esta función recibe como parámetro un conjunto de items y un símbolo, y retorna el conjunto goto(items, symbol). Este método permite darle valor al parámentro just kernel=True para calcular el conjunto de items kernels. En caso contrario, se requiere el conjunto con los firsts de la gramática para entonces calcular la clausura.

Para la construcción del autómata LR(1) se utilizó el siguiente algoritmo:

```
def build_automata(self, g):
    assert len(g.Start_symbol.Productions) == 1, 'Grammar must be augmented'
    firsts = compute_firsts(g)
    firsts[g.Eof] = ContainerSet(g.Eof)
    start_production = g.Start_symbol.Productions[0]
    start_item = Item(start_production, 0, lookaheads=(g.Eof,))
    start = frozenset([start_item])
    closure = self.closure_lr1(start, firsts)
    automata = State(frozenset(closure), True)
    pending = [start]
    visited = {start: automata}
    while pending:
        current = pending.pop()
        current_state = visited[current]
        for symbol in g.Terminals + g.Non_terminals:
            #  (Get/Build `next_state`)
            a = self.goto_lr1(current_state.state, symbol, firsts, True)
            if not a:
                continue
            try:
                next_state = visited[a]
            except KeyError:
                next_state = State(frozenset(self.goto_lr1(current_state.state, symbol, firsts)), True)
                visited[a] = next_state
                pending.append(a)
            current_state.add_transition(symbol.name, next_state)
    automata.set_formatter(multiline_formatter)
    return automata
```

El siguiente algoritmo se utiliza para llenar la tabla Acción-Goto, bajo las siguientes reglas:

- Sea "$X \to \alpha.c\omega, s$" un item del estado I_i y $Goto(I_i, c) = I_j$.
Entonces $ACTION[I_i, c] =$ 'S_j'.

- Sea "$X \to \alpha., s$" un item del estado I_i.
Entonces $ACTION[I_i, s] =$ 'R_k' (producción k es $X \to \alpha$).

- Sea I_i el estado que contiene el item "$S' \to S., \$$" (S' distinguido).
Entonces $ACTION[I_i, \$] =$ 'OK'.

- Sea "$X \to \alpha.Y\omega, s$" item del estado I_i y $Goto(I_i, Y) = I_j$.
Entonces $GOTO[I_i, Y] = j$.