Proyecto de Compilación

Dayron Fernández Acosta
Javier Villar Alonso
Julio José Horta Vázques

28 de noviembre de 2022

1. Gramática

En este trabajo se propone la creación de una gramática y su compilador, tal que permita la simulación de juegos de tableros; centrandonos especificamente en el ajedrez. Para el manejo de este sistema se implementó un lenguaje de dominio específico (DSL, por sus siglas en ingles Domain Specific Language).

El lenguaje que implementamos permite la creación de código de alto nivel parecido al que observamos a lo largo de la industria. En su creación se encuentra grupos de instrucciones como la definición de de variable y funciones, comparativas y ciclos. En este lenguaje los contextos se encontraran encerrados entre llaves , y todos las posibles oraciones serán seguidas de un punto y coma. En total el lenguaje cuenta con alrededor de 80 producciones, 42 no terminales y 60 terminales. La estructura de esta gramática se diseñó teniendo en cuenta la creación de un parser **LR(1)**[1].

1.1. Clases de la Gramática

Para el trabajo e implementacion de la gramática se utilizaron varias clases que facilitaron el trabajo a la hora de la declaracion y resolucion. Para la representacion completa del lenguaje fueron utilizadas 5 clases: *Terminal, NoTerminal, Grammar , Production , Symbol*. Específicamente la clase *Grammar* es generada a partir de una lista de NoTerminales

1.1.1. Symbol

La clase *Symbol* es utilizada como clase abstracta que permite englobar cualquier componente que forme parte de alguna producción de la gramática. Más importante, en esta clase se encuentra el metodo dinámico *is_terminal*, el cual me permite identificar con que tipo de instancia hereditaria se está trabajando. Los métodos *__repr__*[2] son implementados para mostrar de manera mas exacta los valores con los que se trabaja.

Listing 1: Definición de la Clase Symbol

```
class Symbol(metaclass=ABCMeta):
    def __init__(self, name: str) -> None:
        self.name = name
        #self.ast = None

    def is_terminal(self) -> bool:
        return isinstance(self, Terminal)

    def __repr__(self) -> str:
        return self.name

    def __hash__(self) -> int:
        return hash(self.name)
```

[1]Este parser será discutido más adelante.

[2]Este metodo sera redefinido en multiples clases, simpre con el objetivo de realizar el seguimiento del código en ejecución más fácil.

```
    @abstractmethod
    def copy(self):
        pass
```

1.1.2. Terminal

La clase *Terminal* es utilizada para la instanciación de todos los posibles terminales del lenguaje. Para la construción de cada una de estas instancias es necesario el nombre del terminal como *string* y el tipo de *token*[3] que es. Además, esta es heredera de la clase *Symbol* .

Listing 2: Definición de la Clase Terminal

```
class Terminal(Symbol):
    def __init__(self, name: str, value: str = '') -> None:
        super().__init__(name)
        self.value = value if value != '' else name

    def copy(self):
        return Terminal(self.name, self.value)

    def __repr__(self) -> str:
        return f"{self.value}"
```

1.1.3. NoTerminal

Esta clase es la contraparte de la anterior, con ella se instancian todas los *Symbols* que son cabeza de una producción. Esta además de contar la con la propiedad del nombre, mantiene una lista de todos las producciones que genera. Esta última propiedad es ajustada a través de un método, la redefinición de *__iadd__*[4].

Listing 3: Definición de la Clase No Terminal

```
class NonTerminal(Symbol):
    def __init__(self, name: str, prodList: List[Production] = None):
        super().__init__(name)
        self.name = name
        self.productions = prodList if prodList is not None else []
        self._terminals_set: Set = set()

    def __iadd__(self, prod: Production):
        self.productions.append(prod)
        self._terminals_set.update(prod.get_terminals())
        prod.head = self
        return self

    def copy(self):
        return NonTerminal(self.name, self.productions)
```

[3] Los tipos de token son la manera que tenemos de conocer el tipo de valor real del Terminal

[4] +=

1.1.4. Production

La clase *Production* esta simplemente definida como una lista de *Symbols* y una propiedad llamada *head* que es un *NoTerminal*. La lista de elementos es declarada a la hora de instanciar la producción, sin embargo, la cabeza de ella será actualizada a la hora de añadir dicha produción al no terminal correspondiente.

Listing 4: Definición de la Clase Production

```
class Production:

    def __init__(self, symbols: List[Symbol], ast_node_builder=None):
        self.head: NonTerminal = None
        self.symbols: List[Symbol] = symbols
        #self.ast_node_builder = ast_node_builder
        self.pos: int = 0

    def get_terminals(self) -> Set[Terminal]:
        terminals: Set = set()
        for symbol in self.symbols:
            if (isinstance(symbol, Terminal)):
                terminals.add(symbol)
        return terminals

    def is_eps(self) -> bool:
        return len(self.symbols) == 1 and self.symbols[0] == "EPS"
```

Listing 5: Ejemplo de añadir una producció a un no Terminal.

```
let_dec += Production([_Let, all_types, _ID, _Assign, expr])
```

2. Tokenizer

El tokenizer es una pequeña maquina de estados particular para nuestro lenguaje el cual es capaz de leer cualquier token de este. Recibe un string con la cadena en el lenguaje y va iterando por las posiciones de este.

- En caso de encontrarse un alfanumérico entra a la maquina de estados de alfanumérico, comprueba si es una palabra clave o no en un diccionario de palabras clave, y determina que token utilizar entre identificador o alguna palabra reservada del lenguaje.
- En caso de encuentra números entra al estado reconocedor de números el cual determina si son números flotantes o enteros.
- Existe un estado para reconocer strings
- Existe un estado que si recibe un símbolo como + , = , / , - , ¡=, los reconocerá como símbolos del sistema
- Posee un estado para leer comentarios

En caso de estar leyendo un token y un estado se trabe y no sepa a que estado moverse texto aun por reconocer este reconocerá el error.

Todo esto devolverá una cadena de tokens que serán usados en todas las etapas en lo adelante.

3. Parser

El parser implementado es el LR(1) canónico, el cual es un analizador sintáctico LR (k) para $k = 1$, es decir, con un único terminal de búsqueda anticipada. El atributo especial de este analizador es que cualquier gramática LR (k) con $k > 1$ se puede transformar en una gramática LR (1). Para la implementación de este se definieron varias clases, entre ellas la clase `LR1Item` que representa la definición de Item LR(1) e Item SLR, para este último se permite la no entrada del parámetro `lookahead`.

La clase `LR1Item` tiene como atributos una producción, la posición del punto y el terminal lookahead. La posición del punto indica los símbolos que han sido recorridos y los que no, los que han sido recorridos son los símbolos cuyas posiciones son menores que la posición del punto y los que no son los símbolos cuyas posiciones son mayor o igual que la del punto.

Otra de las clases implementadas es la clase `State` que representa un estado del autómata LR(1). El constructor de esta clase recibe como parámetro una lista de items LR1, a partir de los cuales el método `build` de la propia clase construirá el estado inicial. Otro de los métodos implementados en esta es el `set_go_to`, el cual tiene como función calcular todas las transiciones a partir de un conjunto de items y un símbolo.

Para la representación del autómata se implementó una clase con el mismo nombre. El constructor de esta clase recibe como parámetro una gramática. Esta clase al instanciarse crea una lista de estados como atributo de la clase, para llegar a construir esta lista primero se extendiende la gramática mediante el método `extended_grammar` de la propia clase el cual añade una nueva producción con el símbolo inicial como cabeza `''S''` el cual pasará a ser el nuevo comienzo de la gramática. Luego se obtiene una lista de no terminales de la gramática y por cada una de las producciones de este se generan los items iniciales, luego a partir de este, el estado inicial, y del estado inicial los próximos estados para cual se simula una cola utilizando los slices de Python partiendo del estado inicial.

Para el manejo de las clases anteriores y creación de las tablas `action_table` y `go_to_table` se implementó la clase `LR1Table`. El constructor de esta clase recibe como parámetro una gramática a partir de la cual serán construidas las tablas mencionadas. El método `build_table` perteneciente a esta clase es el encargado de la construcción de las tablas, estas están representadas por una lista de diccionarios, donde cada posición i de la lista corresponde a un estado de la clase `Autómata`. Por cada uno de estados son creados dos diccionarios locales, uno de estos almacenará las acciones a realizar y número de estado próximo, dado un símbolo, la acción `SHIFT` se indicará por el caracter inicial de esta acción si el símbolo revisado es un terminal, en caso contrario (No terminal), este es añadido como llave del otro diccionario `go_to`, y el valor correspondiente será el número del estado de la proxima transición.

Luego se procede a analizar los lookaheads, y para esto se define un diccio-

nario de Terminal como llave y lista de items LR1 como valor, luego se chequea si el lookahead está contenido en el diccionario mencionado, en caso de no encontrarse, este es añadido como llave al diccionario de acciones y es asignado la acción `REDUCE` representada por su caracter inicial `''R''` y la producción a la q se deberá reducir.

En caso de ser detectado el Terminal `$` y que la cabeza de la producción sea `S` (No terminal inicial) entonces se almacena, el terminal como llave y string `OK` como valor.

Para fines de optimización el resultado del método `build_table` se registra en un fichero .json, una vez estos almacenados en el directorio del proyecto, son detectados en un nuevo programa y se evita la reconstrucción de dichas tablas cargándolos desde el directorio.

4. Árbol de Sintaxis Abstracta

La construcción del árbol de sintaxis abstracta del lenguaje se basa en la construcción de un Nodo Program. Este nodo tiene como propiedades una lista de nodos de statements secuenciales y son inicializado con valor "nones". A partir de su construcción es evaluada mediante el médodo $\text{build}_a st, basado en crear cada nodo statement que actuará$

Los statemenents que presentados en el DSL son los siguientes:

1. Ciclos : nodo de ciclo usado para representar código que deseemos repetir una cierta cantidad de veces. Esta consistirá en una condición que permitirá ejecutar un cuerpo de programa con su propio contexto derivado del contexto padre en el que se anda trabajando.

2. Declarador de variables y re definidor de variables: La creación consiste en una sintaxis de tipo let que recibirá el tipo, identificador y el valor de la expresión a partir de su nodo constructor. Estos let crearán guardaran sus valores en nuestro contexto creado por el creador de este y usado posteriormente en nuestro programa. El redefinidor utiliza un su propio nodo para cambiar el valor de las variables

3. Declarador y llamadas de funciones: La declaración consiste en un cuerpo de statements creados a base de un identificador con unas entradas argumentos evaluados a partir de la su constructor y guardados en nuestro contexto, mientras que la llamada de funciones permite llamar a estas siempre que se pasen los argumentos correctamente

4. Condicionales : Consiste en preguntas de decisión condicionales que llevan como propiedad una condición, un nodo programa como cuerpo a ejecutar en caso de cumplir la condición, en caso contrario se procederá a procesar y operar su hijo el nodo else.

5. Funciones de diccionarios: Permitimos utilizar diccionarios en el lenguaje que den las funcionalidades básicas de buscar, extraer, guardar y definir elementos en estos. Cada operación tiene su propio nodo

6. Para poder visualizar el tablero permitimos la funcionalidad de mostrar elementos en consola con el comando print el cual tiene su propio nodo dedicado.

7. En esta gramática se manejan las operaciones básicas matemáticas como la suma, resta, multiplicación y división

8. Este lenguaje presenta la implementación de diccionarios para mayor facilidades a la hora de guardar y trabajar información.

9. Como medio de trabajo con nuestro tablero de ajedrez, tenemos un Board para guardar y trabajar las propiedades de tableros de dos dimensiones y piezas. Estas piezas se ubicaran en el tablero y permitirán simular un posición en un tablero y crear jugadas a partir del los movimientos definidos en el lenguaje, por defecto se adicionan los movimientos básicos del ajedrez . Además de contar con un método para borrar una posición del tablero y otro para insertar cada una pieza en el tablero independientemente del movimiento.

Bonus- Este no tiene que ser un lenguaje solo para jugar ajedrez, ya que cada pieza tendrá su tipo de movimiento único que permite la creación de piezas con el movimiento que el usuario o desarrollador decida definir. Para hacer esto a la hora de crear una pieza, justo después de nombrarla y decir su color le pasamos una función de movimiento y esta la adoptara, pudiéndose crear piezas de cualquier juego de tablero

Estructura de un nodo estándar:

Este tiene 4 métodos principales, su constructor, su validador, su transpilador y su evaluador:

1. Constructor: Este se encarga de la inicialización de las propiedades. Si una de sus propiedades depende de sus hijos, ejemplo un nodo suma que tiene dos números que sumar, esta sera guardada como la inicialización de un nodo numérico en su hijo derecho y otro similar en su hijo izquierdo.

2. Validación: Este nodo se usa para comprobar que las propiedades de cada nodo cumplan con las restricciones de hijos esperados. Una de las maneras para hacer esto es usa un contexto que se pasa por cada nodo el cual lleva almacenado cada variable declarada en ese ámbito, así como una lista de funciones globales. Este contexto sabe el tipo de retorno de cada elemento que tiene almacenado por lo que puede comprobar por cada nodo si sus hijos comparten tipo con su valor de retorno o al menos su valor esperado.

3. Transpilador: Esta función se encarga de escribir en código de python todo el programa escrito.

4. Evaluador: Esta función se encarga de una vez todo este construido y validado hacer una ejecución del programa.