Informe - Compiler for BASIC Dartmouth 64

Presentado a:

Ángel Augusto Agudelo Zapata

Presentado por:

Santiago Valencia Díaz

Jesús Daniel Mosquera Herrera

Asignatura:

Compiladores

Universidad Tecnológica de Pereira

2024-1

Tabla de Contenido

Introducción

En el mundo de la programación, los compiladores son cruciales para crear software. Convierten el código legible por el ser humano en código legible por la máquina, haciendo posible que los ordenadores lo entiendan y lo ejecuten y desempeñan un papel importante en todos los software, aplicaciones e interacciones digitales que se tienen.

Un compilador es un programa informático que traduce código fuente escrito en un lenguaje de programación de alto nivel a un código objeto ejecutable o a otro lenguaje de programación de más bajo nivel (como el lenguaje ensamblador o el código de máquina). El propósito más común de compilar código fuente es crear un programa ejecutable, y la sofisticación de un compilador puede influir muchísimo en la eficacia y velocidad del programa ejecutable final.

Entender cómo funcionan los compiladores es fundamental para el desarrollo de software porque tiende un puente entre el código de alto nivel legible por el ser humano y las instrucciones de bajo nivel comprensibles por la máquina.

La construcción de un compilador generalmente implica varias etapas:

1. Análisis Léxico.

2. Análisis Sintáctico.

3. Análisis Semántico.

4. Generación de Código Intermedio.

5. Optimización del Código Intermedio.

6. Generación de Código Objeto.

Estas etapas pueden variar en complejidad dependiendo del diseño del compilador y del lenguaje de programación objetivo. La construcción de un compilador es un proceso complejo que requiere un buen entendimiento de la teoría de compiladores, así como habilidades en programación y matemáticas. Además, la mayoría de los compiladores modernos también incluyen herramientas para depuración, generación de informes de errores y otras características adicionales.

Marco Conceptual

Análisis Léxico:

- El análisis léxico constituye la primera fase, aquí se lee el programa fuente de izquierda a derecha y se agrupa en componentes léxicos (tokens), que son secuencias de caracteres que tienen un significado.
- Se comprueba que los símbolos del lenguaje (palabras clave, operadores, etc.) se han escrito correctamente.
- Como la tarea que realiza el analizador léxico es un caso especial de coincidencia de patrones, se necesitan los métodos de especificación y reconocimiento de patrones. Se usan principalmente los autómatas finitos que aceptan expresiones regulares. Sin embargo, un analizador léxico también es la parte del traductor que maneja la entrada del código fuente, y puesto que esta entrada a menudo involucra un importante gasto de tiempo, el analizador léxico debe funcionar de manera tan eficiente como sea posible.

Análisis Sintáctico:

- Es la etapa del proceso de compilación donde se verifica que la estructura de un programa de computadora sigue las reglas gramaticales del lenguaje de programación en cuestión.
- Se divide en dos fases: análisis léxico y análisis gramatical.
- En el análisis léxico se identifican los tokens individuales del código fuente, mientras que en el análisis gramatical se determina cómo se combinan estos tokens para formar expresiones válidas según la gramática del lenguaje.
- El resultado del análisis sintáctico es generalmente un árbol sintáctico o un grafo de dependencias que representa la estructura del programa.

Árbol de Sintaxis Abstracta (AST):

- Es una estructura de datos que representa la estructura sintáctica de un programa de computadora de una manera más abstracta que un árbol sintáctico completo.

- Elimina detalles gramaticales irrelevantes y se enfoca en la estructura lógica del programa.
- Es comúnmente utilizado como una representación intermedia en el proceso de compilación, ya que es más fácil de manipular y analizar que el código fuente original.
- Cada nodo del AST representa una construcción del lenguaje, como una expresión, una declaración o una función, y sus hijos representan los componentes de esa construcción.

Analizador Semántico:

- Es la etapa del proceso de compilación que se encarga de verificar que el programa tiene sentido desde el punto de vista del significado o la semántica del lenguaje.
- Comprueba que las construcciones del programa no solo sean sintácticamente correctas, sino también semánticamente coherentes.
- Esto implica verificar el uso correcto de variables, tipos de datos, operadores y otras construcciones del lenguaje.
- También puede implicar la resolución de referencias a variables y funciones, y la inferencia de tipos de datos cuando sea posible.

Código Intermedio:

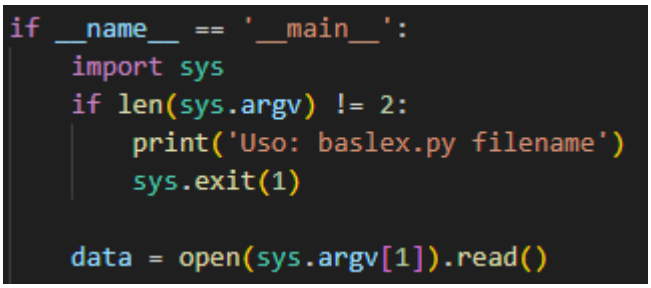
- Es una representación intermedia del programa de computadora que se genera durante el proceso de compilación, entre el análisis del código fuente y la generación de código de máquina.
- Puede tomar varias formas, como código de tres direcciones, código de pila, código de registros, etc.
- El código intermedio es más abstracto que el código de máquina, pero más concreto que el código fuente original.
- Facilita la optimización del programa y la portabilidad del compilador, ya que las optimizaciones y la generación de código final pueden realizarse en función del código intermedio en lugar del código fuente directamente.

Desarrollo

Se pidió el desarrollo de un compilador del lenguaje ‘BASIC’ usando el lenguaje ‘Python’ como fuente. El compilador está basado en la versión más antigua creada por la Universidad de Dartmouth y éste tomó el nombre de ‘Dartmouth BASIC’. A lo largo del semestre, se le añadieron nuevas funcionalidades a esta versión de ‘BASIC’ para poder asemejarse a versiones más modernas y que permitan ejecutar programas de ‘BASIC’ más avanzados y complejos. A continuación, explicaremos nuestros avances en cada fase del compilador:

Analizador léxico

1. **Inicio del Análisis Léxico:**
    - El proceso comienza con la entrega del código fuente del programa BASIC al analizador léxico. Creamos un archivo ‘baslex.py’ y allí se define una función ‘main’ donde se pueda hacer la lectura del contenido de un archivo en formato ‘.bas’.



```
if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Uso: baslex.py filename')
        sys.exit(1)

    data = open(sys.argv[1]).read()
```

    - El analizador léxico es responsable de escanear el código fuente carácter por carácter, reconociendo y clasificando los diferentes elementos léxicos que componen el programa, como palabras clave, identificadores, literales, operadores y símbolos especiales. Usando la librería ‘SLY’, se define una clase ‘Lexer’ que

contendrá la definición de los tokens y expresiones regulares.

```
import sly
import re

class Lexer(sly.Lexer):

    reflags = re.IGNORECASE

    def __init__(self, context):
        self.context = context
```

2. **Reconocimiento de Palabras Clave y Símbolos Especiales:**
    - Se define un diccionario ‘tokens’ dentro de la clase’ Lexer’, donde se almacenan las palabras clave específicas del lenguaje ‘BASIC’, como “LET”, “READ”, “DATA”, "PRINT", "IF", "THEN", "FOR", "NEXT", entre otras. Estas palabras clave tienen significados especiales dentro del lenguaje y son fundamentales para definir la estructura y el comportamiento del programa.

```
tokens = {
    # Keywords
    LET, READ, DATA, PRINT, GOTO, IF,
    THEN, FOR, NEXT, TO, STEP, END,
    STOP, DEF, GOSUB, DIM, REM, RETURN, BLTIN, INPUT, RESTORE,

    # Operadores de relacion
    LT, LE, GT, GE, NE,

    # Identificadores
    IDENT, FNAME,

    # Constantes
    INTEGER, FLOAT, STRING,
    NEWLINE,
}
```

    - Además, el analizador léxico también reconoce símbolos especiales como paréntesis, comas, operadores aritméticos y de comparación, y otros caracteres que tienen un significado específico en el lenguaje.

```
# Literales
literals = '+-*/^=():,;'
```

3. **Identificación de Identificadores, Literales y Tokens:**
    - Los identificadores son nombres dados a variables y funciones dentro del programa. El analizador léxico identifica estos identificadores y los relaciona con su expresión regular.. A su vez, también se relaciona el resto de tokens del lenguaje con su respectiva expresión regular.

```
# Expresiones regulares
@_(r'REM( .*)?')
def REM(self, t):
    t.value = t.value[4:] if len(t.value) > 3 else ''
    return t

LET    = r'LET'
READ   = r'READ'
DATA   = r'DATA'
PRINT  = r'PRINT'
GOTO   = r'GOTO'
IF     = r'IF'
THEN   = r'THEN'
FOR    = r'FOR'
NEXT   = r'NEXT'
TO     = r'TO'
STEP   = r'STEP'
END    = r'END'
STOP   = r'STOP'
DEF    = r'DEF'
GOSUB  = r'GOSUB'
DIM    = r'DIM'
RETURN = r'RETURN'
INPUT = r'INPUT'
RESTORE =r'RESTORE'

BLTIN = r'SIN|COS|TAN|ATN|EXP|ABS|LOG|SQR|RND|INT|TAB|DEG|PI|TIME|LEN|LEFT\$|MID\$|RIGHT\$|CHR\$'

FNAME = r'FN ?[A-Z]'
IDENT = r'[A-Z][A-Z0-9]*\$?'
```

```
NE = r'<>'
LE = r'<='
LT = r'<'
GE = r'>='
GT = r'>'
```

- El analizador también reconoce los números enteros, números de punto flotante y cadenas de caracteres. El analizador léxico reconoce estos literales y se define la expresión regular adecuada para cada uno.

```
@_(r'((\d*\.\d+)(E[\+-]?\d+)?|([1-9]\d*E[\+-]?\d+))')
def FLOAT(self, t):
    t.value = float(t.value)
    return t

@_(r'\d+')
def INTEGER(self, t):
    t.value = int(t.value)
    return t

@_(r'"[^"]*"?')
def STRING(self, t):
    t.value = t.value[1:-1]
    return t
```

4. **Manejo de Espacios en Blanco y Caracteres especiales:**
   - El analizador léxico también maneja espacios en blanco y otros caracteres en el código fuente. Los espacios en blanco, como espacios, tabulaciones y saltos de línea, se utilizan para mejorar la legibilidad del código y también se ignoran durante el análisis.

```
@_(r'\n+')
def NEWLINE(self, t):
    self.lineno += 1
    t.value = t.value.replace('\n', '\'n')
    return t
```

```
# Ignorar
ignore = ' \t\r'
```

5. **Generación de Tokens:**
    - A medida que el analizador léxico identifica cada elemento léxico en el código fuente, genera tokens correspondientes que representan estos elementos. Estos tokens se utilizan como entrada para las etapas posteriores del proceso de compilación, como el análisis sintáctico. Dentro de 'baslex.py', esto se hace en la función 'main', donde se imprime cada token a medida que se lee el código fuente.

```
if __name__ == '__main__':
    import sys
    if len(sys.argv) != 2:
        print('Uso: baslex.py filename')
        sys.exit(1)

    data = open(sys.argv[1]).read()

    lex = Lexer()
    for tok in lex.tokenize(data):
        print(tok)
```

Analizador sintáctico

1. **Inicio del Análisis Sintáctico:**
    - Una vez que el analizador léxico ha generado una secuencia de tokens a partir del código fuente, la tarea del analizador sintáctico es verificar que esta secuencia de tokens sigue la gramática definida por el lenguaje BASIC.
    - El analizador sintáctico utiliza reglas gramaticales y estructuras de datos como gramáticas libres de contexto o diagramas de sintaxis para esta verificación. Se crea el archivo 'basparse.py', y en este archivo se desarrolla el analizador sintáctico, empezando por la creación de la clase 'Parser', además de importar la clase 'Lexer' desde 'baslex.py' y todo el contenido de las estructuras de datos para el árbol AST, del cual se hablará más adelante. Se crea por fuera una clase 'SyntaxError' como método de depuración para las instrucciones de 'BASIC' que contengan errores sintácticos y que permitan mostrar al usuario en dónde se encontró el error durante el proceso de análisis. Dentro de la clase 'Parser', se define la bandera 'expected_shift_reduce', la cual permite ignorar las advertencias que arroja

el analizador sintáctico sobre conflictos shift/reduce que se encontraron durante el análisis. Para nuestro analizador en específico, existen dos conflictos shift/reduce, entonces se le asigna un valor de **2** a la bandera. Para efectos de depuración, se define 'debugfile', el cual escribe todo el proceso de análisis de la gramática con las reglas del lenguaje, los símbolos terminales y no terminales, los estados del autómata, etc, en un archivo de texto con nombre 'parse.txt'. Se importan también los tokens definidos en el diccionario 'tokens' de la clase 'Lexer'.

```python
# basparse.py

from rich import print
import sly

from baslex    import Lexer
from basast    import *

class SyntaxError(Exception):
    pass

class Parser(sly.Parser):

    expected_shift_reduce = 2
    debugfile = 'parse.txt'

    tokens = Lexer.tokens
```

2. **Reconocimiento de la Estructura del Programa:**
    - El analizador sintáctico comienza reconociendo las estructuras principales del programa, como el cuerpo del programa, comandos, declaraciones, bloques de control (como bucles y condicionales), funciones, etc.

```
# Definición de programa y stmt "statement"

@_("program stmt")
def program(self, p):
    line, stmt = p.stmt
    p.program[line] = stmt
    return p.program

@_("stmt")
def program(self, p):
    line, stmt = p.stmt
    return Program({line: stmt})

@_("error")
def program(self, p):
    raise SyntaxError("Malformed Program")

# Definición de comandos

@_("INTEGER command NEWLINE")
def stmt(self, p):
    return (p.INTEGER, p.command)

@_("command")
def command(self, p):
    return p.command
```

- Se definieron reglas gramaticales para determinar cómo se combinan los tokens para formar estas estructuras. Por ejemplo, una declaración PRINT seguida de una lista de expresiones y con un ‘optend’ que puede ser la coma, el punto y coma o simplemente puede quedar vacío, sería reconocida como una instrucción de impresión válida. También hay otras combinaciones válidas que constituyen una instrucción de impresión.

```
# Instrucción PRINT

@_("PRINT plist optend")
def command(self, p):
    return Print(p.plist + [ p.optend ])

@_("PRINT")
def command(self, p):
    return Print([])

@_("sep", "empty")
def optend(self, p):
    return p[0]

@_("PRINT error")
def command(self, p):
    raise SyntaxError("Malformed PRINT instruction")
```

3. **Análisis de la Precedencia:**
    - Dentro del analizador sintáctico, se estableció la jerarquía y precedencia de los operadores. Esto se define al principio de la clase 'Parser'.
    - Por ejemplo, en la expresión `A + B * C`, el analizador sintáctico debe reconocer que la multiplicación tiene una precedencia más alta que la suma y agrupar los tokens en consecuencia.

```
precedence = (
    ('left', '+', '-'),
    ('left', '*', '/'),
    ('left', '^'),
    ('right', UMINUS),
)
```

4. **Manejo de ítems:**
    - El analizador sintáctico debe ser capaz de manejar elementos o ítems de vital importancia y de este modo, construir listas usando estos elementos para ciertas instrucciones como 'PRINT', 'DIM', 'READ', etc, que usen listas. Normalmente se construyen dichas listas usando variables, dimensiones

(arreglos en BASIC), expresiones, etc. Se definieron de la siguiente manera para la instrucción 'DIM', por ejemplo.

```
# Elementos para la instrucción DIM

@_("IDENT '(' expr ')'")
def dimitem(self, p):
    return Variable(p.IDENT, p.expr)

@_("IDENT '(' expr ',' expr ')'")
def dimitem(self, p):
    return Variable(p.IDENT, p.expr0, p.expr1)

@_("dimitem")
def dimlist(self, p):
    return [ p.dimitem ]

@_("dimlist ',' dimitem")
def dimlist(self, p):
    p.dimlist.append(p.dimitem)
    return p.dimlist
```

5. **Identificación de Errores Sintácticos:**
    - Si el analizador sintáctico encuentra una secuencia de tokens que no sigue las reglas gramaticales del lenguaje, genera un mensaje de error sintáctico.
    - Este mensaje puede incluir información sobre la ubicación y la naturaleza del error, lo que ayuda al programador a corregirlo.

```
def error(self, p):
    lineno = p.lineno if p else 'EOF'
    value  = p.value  if p else 'EOF'
    if self.context:
        self.context.error(lineno, f"Syntax Error: {value}")
    else:
        print(f"Syntax Error: {value} at line {lineno}")

def __init__(self, context = None):
    self.context = context
```

6. **Construcción del Árbol de Sintaxis Abstracta (AST):**
    - A medida que el analizador sintáctico verifica la estructura del programa, se construye un árbol de sintaxis abstracta (AST) que representa la estructura jerárquica del programa. Esto se hizo utilizando estructuras de tipo ‘dataclass’ y se define el tipo de dato para cada elemento del nodo.
    - Cada nodo del AST representa una construcción del lenguaje, como una declaración, una expresión o una estructura de control, y sus hijos representan los componentes de esa construcción.

AST (Árbol de sintaxis Abstracta)

Construcción del AST:

- Después de que el analizador sintáctico ha verificado la estructura del programa y ha reconocido las diversas construcciones del lenguaje, como declaraciones, expresiones y estructuras de control, se construye el AST. Para esto, se creó el archivo ‘basast.py’, y se importaron diferentes librerías útiles para esta tarea. A su vez, también se define el patrón Visitor, el cual es ampliamente utilizado en intérpretes, compiladores y procesadores de lenguajes, en general, y su propósito es separar algoritmos de los objetos sobre los que operan, algo de vital importancia para el manejo del árbol AST. Como primer paso, se define la clase ‘Visitor’, la cual define una interfaz de visitante que se utiliza para recorrer el AST y utiliza la metaclase ‘multimeta’ para soportar métodos multimétodo. La clase Node es la clase base para todos los nodos del AST y éste incluye un método ‘accept’ que acepta un visitante y permite que el visitante realice operaciones en el nodo. Posteriormente, se crean dos clases adicionales que heredan de ‘Node’, las cuales son ‘Statement’ y ‘Expression’, y sirven como clases base para los nodos que representan declaraciones y expresiones.

```python
# basast.py

from dataclasses import dataclass, field
from multimethod import multimeta
from typing import List, Dict, Optional

# ----------------------------------------------------------------------
# Definicion Estructura del AST
# ----------------------------------------------------------------------
class Visitor(metaclass=multimeta):
    pass

@dataclass
class Node:
    def accept(self, v:Visitor, *args, **kwargs):
        return v.visit(self, *args, **kwargs)

@dataclass
class Statement(Node):
    pass

@dataclass
class Expression(Node):
    pass
```

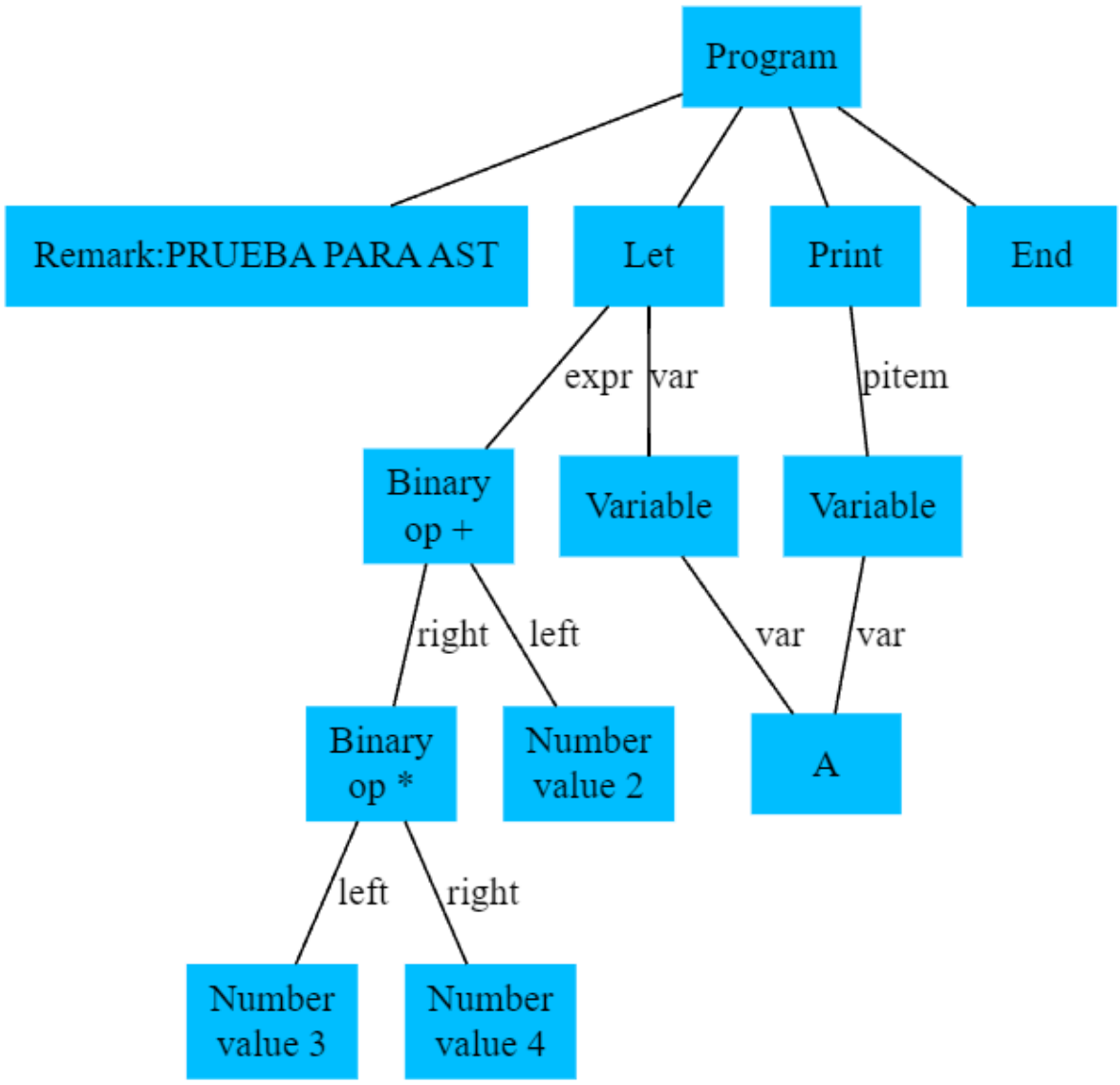
- Durante el análisis sintáctico, cada regla de producción puede estar asociada con la creación de nodos en el AST para representar la estructura del programa. Por ejemplo, una regla de producción para una declaración “PRINT” podría crear un nodo "Print” en el AST, como se puede ver a continuación.

```python
@dataclass
class Print(Statement):
    plist: List[Expression]
    optend: str = None
```

Representación Jerárquica:

- El AST organiza las construcciones del programa en una estructura jerárquica donde cada nodo representa una construcción del lenguaje y sus hijos representan los componentes de esa construcción.

- Por ejemplo, en una expresión aritmética como "2 + 3 * 4", el nodo raíz representaría la operación de suma, con dos nodos hijos que representan los operandos (2 y el resultado de la multiplicación 3 * 4).



Análisis Semántico e Intérprete

En esta etapa del compilador, una vez realizado el análisis sintáctico y se haya construido el árbol AST de un programa de 'BASIC', se procede a hacer el análisis semántico. Se revisa el programa fuente para tratar de encontrar errores semánticos y asegurarse que todo esté en orden para poder entrar a la fase de interpretación de código. Estas son las diferentes directrices definidas para esta fase con su respectivo desarrollo:

1. **Añadido de funciones predefinidas:**
    - Se definen funciones que se encuentran en el manual (SIN, COS, TAN, ATN, EXP, ABS, LOG, SQR, RND, INT) y otras que fueron añadidas en versiones modernas de 'BASIC', como las funciones TAB y TIME, dentro de un diccionario 'self.functions'. Se inicializa previamente la clase 'Interpreter' y se inicializa dentro del mismo el diccionario de funciones que podrá utilizar el intérprete.

```
class Interpreter(Visitor):
    def __init__(self, prog, verbose = False):
        self.prog = prog
        self.verbose = verbose
        self.dc = 0
        self.start_time = time.time() # Capturar el tiempo desde que inició el intérprete

        # Diccionario de funciones predefinidas
        self.functions = {
            'SIN'   : lambda x: math.sin(x),
            'COS'   : lambda x: math.cos(x),
            'TAN'   : lambda x: math.tan(x),
            'ATN'   : lambda x: math.atan(x),
            'EXP'   : lambda x: math.exp(x),
            'ABS'   : lambda x: abs(x),
            'LOG'   : lambda x: math.log(x),
            'SQR'   : lambda x: math.sqrt(x),
            'INT'   : lambda x: int(x),
            'RND'   : lambda x: random.random(),
            'TAB'   : lambda x: ' '*x,
            'DEG'   : lambda x: x * (180.0/3.141592654),
            'PI'    : self.return_pi,
            'TIME'  : self.get_time,
            'LEN'   : self.len_str,
            'LEFT$' : lambda x,n  : x[:n],
            'RIGHT$': lambda x,n : x[-n:],
```

```
        'sin'    : lambda x: math.sin(x),
        'cos'    : lambda x: math.cos(x),
        'tan'    : lambda x: math.tan(x),
        'atn'    : lambda x: math.atan(x),
        'exp'    : lambda x: math.exp(x),
        'abs'    : lambda x: abs(x),
        'log'    : lambda x: math.log(x),
        'sqr'    : lambda x: math.sqrt(x),
        'int'    : lambda x: int(x),
        'rnd'    : lambda x: random.random(),
        'tab'    : lambda x: ' '*x,
        'deg'    : lambda x: x * (180.0/3.141592654),
        'pi'     : self.return_pi,
        'time'   : self.get_time,
        'len'    : self.len_str,
        'left$'  : lambda x,n  : x[:n],
        'right$': lambda x,n : x[-n:]
    }
```

2. **Añadido del comando ‘INPUT’ para entradas por teclado:**
    - Este es un comando que no se encontraba disponible en Dartmouth BASIC, y fue introducido en ‘BASIC V2’, o mejor conocido como Commodore BASIC, por lo que no se encontraba en nuestro analizador léxico y sintáctico anteriormente. Se añadió el token ‘INPUT’ a nuestro analizador léxico para luego poder procesar dicho token en nuestro analizador sintáctico, basándose en la siguiente regla: inputStatement := (label | ";" | ",")* variable ("," variable)*. También se creó su respectivo nodo dentro del árbol AST.

```
INPUT = r'INPUT'
```

```
# Instrucción INPUT

@_("INPUT [ STRING sep ] varlist")
def command(self, p):
    return Input((p.STRING, p.sep), p.varlist)

@_("INPUT error")
def command(self, p):
    raise SyntaxError("Malformed INPUT instruction")
```

```
@dataclass
class Input(Statement):
    label: str
    vlist: List[Expression]
```

- Dentro del intérprete, se definió su ejecución de la siguiente manera, usando el patrón Visitor. El mismo patrón se usa para el resto de instrucciones de BASIC.

```
def visit(self, instr: Input):
    label = instr.label
    # Asegurarse de que 'label' es un string
    if isinstance(label, tuple):
    # Convertir la tupla en un string, uniendo sus elementos con un separador
        label = ' '.join(str(item) for item in label if item is not None)

    if label:
        # Escribir mensaje antes de solicitar la entrada de datos
        label = label.rstrip(';').strip()
        label = label.rstrip(',').strip()
        sys.stdout.write(label)

    for variable in instr.vlist:
        value = input()
        if variable.var[-1] == '$':
            value = String(value)
        else:
            try:
                value = Number(int(value))
            except ValueError:
                value = Number(float(value))
        self.assign(variable, value)
```

3. **Añadir el tipo de datos STRING y sobrecargar el signo + para concatenar:**
    - Este es un requerimiento importante para poder manipular cadenas de caracteres en BASIC. Para este requerimiento, se definió una regla gramatical dentro del analizador sintáctico para poder aceptar cadenas de caracteres como expresiones, como se puede ver a continuación.

```
@_("STRING")
def expr(self, p):
    return String(p.STRING)
```

    - Una vez hechas estas modificaciones en el analizador léxico y sintáctico, respectivamente, se procedió a crear un método visitante para los nodos 'Binary' y 'Logical' en el intérprete. Allí, comprobando que ambos operadores son cadenas, se puede retornar la concatenación si el operador es la suma '+'.

```
def visit(self, instr: Union[Binary, Logical]):
    left = instr.left.accept(self)
    right = instr.right.accept(self)

    # Comparación entre strings
    if isinstance(left, str) and isinstance(right, str):
        if instr.op == '+':
            return left + right
        elif instr.op == '=':
            return left == right
        elif instr.op == '<>':
            return left != right
        elif instr.op == '<':
            return left < right
        elif instr.op == '<=':
            return left <= right
        elif instr.op == '>':
            return left > right
        elif instr.op == '>=':
            return left >= right
        else:
            self.error(f"Incorrect operator {instr.op}")
```

4. **Ignorar las mayúsculas:**
    - Para ignorar las mayúsculas y permitir la escritura de instrucciones en minúsculas, se activa una bandera en el analizador léxico para que esto pueda darse. Inicialmente, esta bandera no existía en nuestro analizador léxico, dado que todo se escribía principalmente en mayúscula para ser lo más fiel posible al código de BASIC original. Sin embargo, otras versiones de BASIC tienen esta característica, por lo que se decidió agregarla. Para que este analizador pueda tener en cuenta los tokens escritos en minúsculas, se añade la librería 'RE' y se establece la bandera 'reflags = re.IGNORECASE', sin tener que realizar más modificaciones en las otras fases del compilador.

```
import sly
import re

class Lexer(sly.Lexer):

    reflags = re.IGNORECASE

    def __init__(self, context):
        self.context = context
```

5. **Verificación de los identificadores**

- Como parte del análisis semántico, se verifica que los identificadores LET / FOR / READ / DIM / INPUT, etc, estén correctamente definidos dentro del intérprete usando el patrón Visitor, muy similar a la implementación del árbol AST anteriormente. Se implementaron dichos métodos visitantes, y de esta manera, también el comportamiento de estas instrucciones dentro del intérprete y cómo se ejecutarán, en consecuencia. Estos son algunos ejemplos.

```
# Patrón Visitor para las instrucciones de BASIC64
def visit(self, instr: Let):
    var = instr.var
    value = instr.expr
    self.assign(var, value)
```

```python
def visit(self, instr: For):
    loopvar = instr.ident
    initval = instr.low
    finval  = instr.top
    stepval = instr.step

    # Si no hay un valor de salto especificado, establecer un valor de 1 por defecto
    if stepval is None:
        stepval = Number(1)

    # Evaluar el valor de salto si existe
    stepval = stepval.accept(self)

    # Verificar si es un loop nuevo
    if not self.loops or self.loops[-1][0] != self.pc:
        # Parece ser un nuevo loop. Hacer la asignación inicial
        newvalue = initval
        self.assign(loopvar, initval)
        self.loops.append((self.pc, stepval))
    else:
        # Es la repetición de un ciclo anterior
        # Actualizar el valor de la variable del loop según el salto
        stepval = Number(self.loops[-1][1])
        newvalue = Binary('+', loopvar, stepval)

        if self.loops[-1][1] < 0:
            relop = '>='
        else:
            relop = '<='

        if not _is_truthy(Logical(relop, newvalue, finval).accept(self)):
            # El ciclo se ha completado. Saltar a la instrucción NEXT
            self.pc = self.loopend[self.pc]
            self.loops.pop()
        else:
            self.assign(loopvar, newvalue)
```

```python
def visit(self, instr: Read):
    for target in instr.varlist:
        if self.dc >= len(self.data):
            # No hay más datos para procesar. El programa termina de ejecutarse
            raise BasicExit()
        # Inicializar 'value' con un valor por defecto antes de usarlo
        value = self.data[self.dc]  # Get the current data item

        if isinstance(target.var, str) and target.var[-1] == '$':
            if self.slicing:
                self.error(f"Cannot proceed with READ instruction at line {self.stat[self.pc]}. String slicing might be enabled.")
            # Si es una variable de tipo string ($), debe obtener una string
            value = value if isinstance(value, str) else str(value)  # Asegurarse de que sea una string
        else:
            # Si es una variable numérica, obtener el tipo correcto. Puede dar un mensaje de error si se activa el corte de cadena desde el compilador
            try:
                value = float(value)  # Convertir a float
            except ValueError:
                self.error(f"The value {value} could not be read.")
        self.assign(target, value)
        self.dc += 1
```

```
def visit(self, instr: Dim):
    for item in instr.dimlist:
        if isinstance(item, Variable):
            vname = item.var
            dim1 = item.dim1
            dim2 = item.dim2

            if not dim2:
                # Variable de una dimensión
                x = dim1.accept(self)
                self.lists[vname] = [0] * x
            else:
                # Variable de doble dimensión
                x = dim1.accept(self)
                y = dim2.accept(self)
                temp = [0] * y
                v = []
                for i in range(x):
                    v.append(temp[:])
                self.tables[vname] = v
```

6. **Verificación de la instrucción ‘END’:**

    - Un programa en BASIC solo debe contener una instrucción END y debe de estar en la última línea.
    - Dentro del intérprete, se implementó un chequeo donde se recorre la lista de instrucciones del programa y se compara si la instrucción entregada es de tipo ‘End’ y si la línea de código recorrida es la última del programa. De lo contrario, el chequeo fallará si no es la última o si definitivamente no existe una instrucción ‘End’ dentro del programa.

```
def check_end(self):
    '''
    Un programa en BASIC solo debe contener una instrucción END
    y esta debe de estar en la ultima linea.
    '''
    has_end = False

    for lineno in self.stat:
        if isinstance(self.prog[lineno], End) and not has_end:
            has_end = lineno

    if not has_end:
        self.error("No hay instrucción END")

    if has_end != lineno:
        self.error("END no es la última instrucción")
```

7. **Verificación de las instrucciones FOR/NEXT:**

   - Este chequeo existe para verificar que haya correlación entre las instrucciones ‘FOR’ y ‘NEXT’ dentro del programa, revisando si ambas comparten la misma variable de iteración. De lo contrario, esto lanzará un mensaje de error, indicando que la instrucción ‘FOR’ no posee su propio ‘NEXT’.

```
def check_loops(self):
    for pc in range(len(self.stat)):
        lineno = self.stat[pc]
        if isinstance(self.prog[lineno], For):
            forinst = self.prog[lineno]
            loopvar = forinst.ident
            for i in range(pc + 1, len(self.stat)):
                if isinstance(self.prog[self.stat[i]], Next):
                    nextvar = self.prog[self.stat[i]].ident
                    if nextvar != loopvar:
                        continue
                    self.loopend[pc] = i
                    break
            else:
                self.error("Instrucción FOR sin NEXT en la linea %s" % self.stat[pc])
```

8. **Verificación de las instrucciones READ/DATA:**

   - Este chequeo existe para poder precargar los datos que se encuentren en una instrucción 'DATA' y que estos, en consecuencia, puedan ser tomados por la instrucción 'READ'. Se implementó un contador de datos para que la instrucción 'READ' tenga mejor control sobre los datos que está recogiendo, y el contador excede la longitud de la lista de datos, se da a entender que no hay más datos por leer y el programa de BASIC terminará su ejecución, tal como está definido igualmente en el método visitante de 'READ'.

```
def collect_data(self):
    '''
    Organizar los datos en la instrucción DATA dentro de una list
    Revisar las instrucciones READ / DATA
    '''

    self.data = []
    for lineno in self.stat:
        if isinstance(self.prog[lineno], Data):
            # Process each item in the mixed list
            for item in self.prog[lineno].mixedlist:
                if isinstance(item, str):
                    # If it's a string, add it directly to the data
                    self.data.append(item)
                else:
                    # If it's an AST node, call accept to get its value
                    self.data.append(item.accept(self))
    self.dc = 0
```

9. **Requerimientos adicionales**

   - Una vez completado el intérprete, se solicitó el uso de dos archivos adicionales, 'bascontext.py' y 'basic.py'. El archivo 'bascontext.py' es una clase de alto nivel que contiene todo lo relacionado con el análisis/ejecución de un programa en BASIC y sirve como depósito de información sobre el programa, incluido el código fuente, informes de errores, etc. El archivo 'basic.py' es un compilador con distintas opciones disponibles para depuración, volcado de análisis léxico/construcción de árbol AST,

modificación de parámetros del intérprete, etc. Ambos archivos trabajan entre sí, por lo que es importante mantener la relación entre ambos. Se solicitó que se agregaran diferentes directrices al compilador, tales como imprimir la versión del compilador, convertir las entradas del usuario a mayúsculas, modificar el índice mínimo de las dimensiones, activar el corte de cadena, etc. Se implementaron los argumentos en el compilador y éstos se transfirieron a 'bascontext.py', para que puedan ser manejados desde allí y puedan transferirse al intérprete, y desde allí, configurar dichos argumentos para que hagan su respectiva labor.

```
# basic.py

'''
Usage: basic.py [-h] [-a style] [-o OUT] [-l] [-D] [-p] [-I] [--sym] [-S] [-R] [-u] [-ar] [-sl] [-n] [-g] [-t] [--tabs] input

Compiler for BASIC DARTMOUTH 64

Positional arguments:
  input             BASIC program file to compile

Optional arguments:
  -h, --help                            Show this help message and exit
  -D, --debug                           Generate assembly with extra information (for debugging purposes)
  -o OUT, --out OUT                     File name to store generated executable
  -l, --lex                             Store output of lexer
  -a STYLE                              Generate AST graph as DOT or TXT format
  -I, --ir                              Dump the generated Intermediate representation
  --sym                                 Dump the symbol table
  -S, --asm                             Store the generated assembly file
  -R, --exec                            Execute the generated program
  -v, --version                         Show the version of the BASIC interpreter
  -u, --uppercase                       Convert all entries to uppercase
  -ar INT, --array-base INT             Set the minimum index of the dimensional arrays (default is 1)
  -sl, --slicing                        Enable string slicing (disable string arrays)
  -n, --no-run                          Don't run the program after parsing
  -g, --go-next                         If no branch from a GOTO instruction exists, go to the next line
  -t, --trace                           Activate tracing to print line numbers during execution
  --tabs INT                            Set the number of spaces for comma-separated elements (default is 15)
  -rn INT, --random INT                 Set the seed for the random number generator
  -p, --print-stats                     Print statistics on program termination
  -w, --write-stats                     Write statistics to a file on program termination
  -of, --output-file                    Redirect PRINT output to a file
  -if INPUT_FILE, --input-file INPUT_FILE  Redirect INPUT to a file
'''
```

```python
from contextlib import redirect_stdout
from rich       import print
from bascontext import Context

import argparse


def parse_args():
  cli = argparse.ArgumentParser(
    prog='basic.py',
    description='Compiler for BASIC programs'
  )

  cli.add_argument(
    '-v', '--version',
    action='version',
    version='0.4')

  fgroup = cli.add_argument_group('Formatting options')

  fgroup.add_argument(
    'input',
    type=str,
    nargs='?',
    help='BASIC program file to compile')

  mutex = fgroup.add_mutually_exclusive_group()

  mutex.add_argument(
    '-l', '--lex',
    action='store_true',
    default=False,
    help='Store output of lexer')

  mutex.add_argument(
    '-a', '--ast',
    action='store',
    dest='style',
    choices=['dot', 'txt'],
    help='Generate AST graph as DOT or TXT format')

  mutex.add_argument(
    '--sym',
    action='store_true',
    help='Dump the symbol table')
```

```
cli.add_argument(
  '-u', '--uppercase',
  action='store_true',
  default=False,
  help='Convert all entries to uppercase')

cli.add_argument(
  '-ar', '--array-base',
  type=int,
  default=1,
  help='Set the minimum index of the arrays (default is 1)')

cli.add_argument(
 '-sl', '--slicing',
  action='store_true',
  default=False,
  help='Enable string slicing (disable string arrays)')

cli.add_argument(
  '-n', '--no-run',
  action='store_true',
  default=False,
  help='Do not run the program after parsing')

cli.add_argument(
  '-g', '--go-next',
  action='store_true',
  default=False,
  help='If no branch from a GOTO instruction exists, go to the next line')

cli.add_argument(
  '-t', '--trace',
  action='store_true',
  default=False,
  help='Activate tracing to print line numbers during execution')

cli.add_argument(
  '--tabs',
  type=int,
  default=15,
  help='Set the number of spaces for comma-separated elements (default is 15)')
```

```
cli.add_argument(
  '-rn', '--random',
  type=int,
  help='Set the seed for the random number generator')

cli.add_argument(
  '-p', '--print-stats',
  action='store_true',
  default=False,
  help='Print statistics on program termination')

cli.add_argument(
  '-w', '--write-stats',
  action='store_true',
  default=False,
  help='Write statistics to a file on program termination')

cli.add_argument(
  '-of', '--output-file',
  action='store_true',
  default=False,
  help='Redirect PRINT output to a file')

cli.add_argument(
  '-if', '--input-file',
  type=str,
  help='Redirect INPUT to a file')

return cli.parse_args()
```

```python
if __name__ == '__main__':

  args = parse_args()
  context = Context()

  if args.input: fname = args.input

  with open(fname, encoding='utf-8') as file:
    source = file.read()

  if args.lex:
    flex = fname.split('.')[0] + '.lex'
    print(f'Printing Lexer output: {flex}')
    with open(flex, 'w', encoding='utf-8') as fout:
      with redirect_stdout(fout):
        context.print_tokens(source)

  elif args.style:
    base = fname.split('.')[0]
    fast = base + '.' + args.style
    print(f'Printing the AST graph: {fast}')
    with open(fast, 'w') as fout:
      with redirect_stdout(fout):
        context.print_ast(source, fast, args.style)

  elif args.sym:
    base = fname.split('.')[0]
    fsym = base + '_symtab.txt'
    print(f'Dumping symbol table: {fsym}')

  else:
    context.parse(source)
    if not args.no_run:
      context.run(args.uppercase, args.array_base, args.slicing, args.go_next, args.trace, args.tabs, args.random, fname, args.print_stats, args.write_stats, args.output
```

```python
class Context:
  def __init__(self):
    self.lexer  = Lexer(self)
    self.parser = Parser(self)
    self.interp = Interpreter(self)
    self.source = ''
    self.ast = None
    self.have_errors = False

  def print_tokens(self, source):
    # Tokenize the source
    tokens = list(self.lexer.tokenize(source))  # Convert to list for iteration
    # Print each token with its details
    for token in tokens:
      # You might want to customize what information to display
      print(f"Token(type={token.type}, value={token.value}, position={token.lineno}:{token.index})")

  def parse(self, source):
    self.have_errors = False
    self.source = source
    self.ast = self.parser.parse(self.lexer.tokenize(self.source))

  def print_ast(self, source, fast, style):
    self.source = source
    self.ast = self.parser.parse(self.lexer.tokenize(self.source))
    dot = DotRender.render(self.ast)
    if style == 'dot':
      with open(fast, "w") as fout:
        fout.write(str(dot))
    elif style == 'txt':
      print(dot)

  def run(self, uppercase, array_base, slicing, go_next, trace, tabs, random_seed, fname, print_stats, write_stats, output_file, input_file):
    if not self.have_errors:
      if output_file:
        base = fname.split('.')[0]
        fprint = base + '_print.txt'
        print(f'Redirecting PRINT output to file: {fprint}')
        if input_file:
          print(f'Redirecting INPUT to read from file: {input_file}')
        with open(fprint, 'w', encoding='utf-8') as fout:
          with redirect_stdout(fout):
            return self.interp.interpret(self.ast.lines, verbose=False, uppercase = uppercase, array_base = array_base, slicing = slicing, go_next = go_next, trace = trace, t
      elif input_file:
        print(f'Redirecting INPUT to read from file: {input_file}')
      return self.interp.interpret(self.ast.lines, verbose=False, uppercase = uppercase, array_base = array_base, slicing = slicing, go_next = go_next, trace = trace, tabs =
```

Código Intermedio

Definición de la Estructura de la Pila:

- Antes de comenzar, es necesario definir la estructura de la pila que se utilizará para representar el código intermedio. En este enfoque, la pila actuará como un área de trabajo donde se realizarán las operaciones intermedias durante la compilación.

Generación de Código Intermedio:

- Durante el análisis sintáctico del programa BASIC, cada vez que se encuentra una construcción que requiere generación de código, se generan instrucciones correspondientes y se colocan en la pila.
- Por ejemplo, cuando se encuentra una expresión aritmética como "2 + 3 * 4", se generan instrucciones para realizar la multiplicación y la suma, y estas instrucciones se colocan en la pila en el orden correcto para garantizar que se evalúen correctamente.
- Se utilizó el árbol AST generado por el analizador sintáctico para poder hacer las relaciones entre las instrucciones de BASIC y las instrucciones intermedias. Se crearon métodos visitantes para ciertas instrucciones, dado que se requiere un análisis más profundo para instrucciones más complejas y determinar cuáles son las instrucciones más adecuadas que pueden replicar el funcionamiento del programa en BASIC, por ejemplo.

Manejo de Operadores y Operandos:

- Los operadores y operandos se manejan de manera similar a como se manejaría en una máquina virtual o en la evaluación de expresiones posfija. Los operandos se colocan en la pila mientras que los operadores se utilizan para realizar operaciones en los operandos que están en la cima de la pila.
- Por ejemplo, para la expresión "2 + 3 * 4", primero se colocaría 2 en la pila, luego 3 y finalmente 4. Después, se realizaría la multiplicación de 3 y 4, y el resultado se colocaría en la pila. Finalmente, se sumaría 2 con el resultado de la multiplicación y el resultado final estaría en la cima de la pila.

Manejando las instrucciones Intermedias:

- Cada operación en la pila puede representarse como una instrucción intermedia. Por ejemplo, la instrucción "ADDI" puede representar una suma entre enteros, la instrucción "MULI" puede representar una multiplicación entre enteros, etc.
- Estas instrucciones intermedias son independientes del hardware específico y del lenguaje de programación de destino, lo que permite una mayor portabilidad del compilador.