Compiler Documentation

Link github: https://github.com/914-Mathe-Andrei/lftc/tree/main/lab3

I. Introduction

The entire implementation is written in python v3.11.

The current compiler contains a lexer that can lexically analyse the source code and produce two symbols table (one for identifiers and one for literals) and a program internal form. The two symbol table use as a data structure a hash table.

II. Details

hash_table.py

The module defines a general purpose *HashTable* class that implements the hash table data structure and its associated operations. The used collision resolution technique is separate chaining. This modules takes advantage of the python's magic methods like *__setitem__()*, *__getitem__()*, *__delitem__()* and *__contains__()* to implement the CRUD operations. The hash function of the data structure uses the built-in hash function implemented by the python's *object* class applied over the key modulo the size of the hash table's array. The class also overrides the *__str__()* function to print and debug the hash table with ease.

The error handling is implemented in the *HashTable* class.

▼ Interface

```
def __init__(self, size: int = 1000):
    """ Initializes the hash table with array of size `size`.

    Args:
        size (int): the size of the array
    """

@property
def entries(self) -> list["HashTable.Entry"]:
    """ Returns a list of all entries in the hash table. """

def __setitem__(self, key: Any, value: Any) -> None:
    """ Adds a new entry in hash table if key does not exist, otherwise it updates the value.

    Args:
        key (Any): key of the entry to be added / updated
        value (Any): value of the entry to be added / updated
    """

def __getitem__(self, key: Any) -> Any:
    """ Return the value associated with the key. If the key does not exist, it raises a KeyError.

    Args:
        key (Any): key in the hash table

    Returns: value associated with the key
```

```
    Raises:
        KeyError: the key does not exist in the hash table
    """

def __delitem__(self, key: Any) -> None:
    """ Deleted the entry associated with the key. If the key does not exist, it raises a KeyError.

    Args:
        key (Any): key of the entry to be deleted

    Raises:
        KeyError: the key does not exist in the hash table
    """

def __contains__(self, key: Any) -> bool:
    """ Checks if key is in the hash table.

    Args:
        key (Any): key to be checked

    Returns: True if key is in hash table, otherwise False
    """

def __len__(self):
    """ Returns the number of entries. """

def __str__(self):
    """ Returns the hash table's data in a json formatted string. """
```

symtable.py

The module defines the two symbol tables for identifiers, *IdSymTable*, and literals, *LiteralSymTable*. The symbols tables are similar, they both use the *HashTable* class as a data structure and implement the add(), *delete()* and *contains()* operations. There is an additional method for retrieving all their entries.

The two symbol tables have the *__str__()* method overridden and implement error handling.

▼ IdSymTable Interface

```
@property
def ids(self) -> list[str]:
    """ Returns a list of all the ids in the table. """

def add(self, id: str) -> None:
    """ Add a new identifier.

    Args:
        id (str): id

    Raises:
        SymTableError: if `id` already exists in the table
    """

def delete(self, id: str) -> None:
    """ Removes an id.

    Args:
        id (str): id
```

```
    Raises:
        SymTableError: if `id` does not exist in the table
    """

def contains(self, id: str) -> bool:
    """ Checks if an id is in the table.

    Args:
        id (str): id

    Returns: True if `id` exists else False
    """

def __str__(self):
    """ Returns a json formatted string as a representation. """
```

▼ LiteralSymTable Interface

```
@property
def literals(self) -> list[str]:
    """ Returns a list of all the literals in the table. """

def add(self, literal: Any) -> None:
    """ Adds a new literal.

    Args:
        literal (Any): literal

    Raises:
        SymTableError: if `literal` already exists in the table
    """

def delete(self, literal: Any) -> None:
    """ Removes a literal.

    Args:
        literal (Any): literal

    Raises:
        SymTableError: if `literal` does not exist in the table
    """

def contains(self, literal: Any) -> bool:
    """ Checks if a literal is in the table.

    Args:
        literal (Any): literal

    Returns: True if `literal` exists else False
    """

def __str__(self):
    """ Returns a json formatted string as a representation. """
```

pif.py

The module contains an enum for token classification (identifier. literal, operator, separator, keyword) and the program internal form defined as a class. The PIF stores pairs of token and token types.

▼ TokenType Enum

```
class TokenType(Enum):
    """ Classes of tokens. """
    IDENTIFIER = 1
    LITERAL = 2
    OPERATOR = 3
    SEPARATOR = 4
    KEYWORD = 5
```

▼ ProgramInternalForm Interface

```
@property
def pairs(self) -> list["ProgramInternalForm.Entry"]:
    """ Returns a list of pif pairs. """

def add(self, token: str, type: TokenType) -> None:
    """ Adds a token pair.

    Args:
        token (str): token
        type (TokenType): type of token (id / literal / operator / separator / keyword)
    """

def __str__(self):
    """ Returns a string containing all the pairs separated by newline. """
```

program.py

The module defines the *Program* class, a representation of a program. It encapsulate the program internal form and the two symbol tables.

errors.py

The module defines types of compiler errors: *SymTableError*, *LexicalError*, *InvalidSymbol*, *InvalidSequence*.

The *LexicalError*, *InvalidSymbol*, *InvalidSequence* classes store the line and column number where the error occurred. Additionaly, the *InvalidSymbol* class stores the symbol that cause the error to be raised.

buffer.py

The module defines the *ImmutableBuffer* class, an immutable buffer. When created, the buffer is initialized with data. The data can be read and put back, but it cannot be altered. A cursor is used for implementing the read and write operations in order to preserve the data. The class also keeps track of the line and column number where the cursor is currently at.

The immutable buffer is useful for the lexical analysis stage when the source code is tokenized.

▼ Interface

```
@property
def line(self) -> int:
    """ Returns the line number where the cursor is currently at.

    Returns: the line number
    """

@property
def col(self) -> int:
    """ Returns the column number where the cursor is currently at.

    Returns: the column number
    """

def get(self, size: int | None = None) -> str:
    """ Returns the next `size` characters in a string.

    If `size` is None than it will read all the remaining characters.

    Args:
        size (int | None): number of characters to read

    Returns: the read characters

    Raises:
        ValueError: if `size` is a negative number
    """

def put(self, size: int | None = None) -> None:
    """ Puts back `size` characters in the buffer. It actually moves the cursor `size` characters back.

    Args:
        size (int | None): number of characters

    Raises:
        ValueError: if `size` is a negative number
    """

def __len__(self):
    """ Returns the length of the buffer. """
```

grammar.py

The module contains a handful of constants that make up a part of the grammar. These constants are used to easily scan and parse the source code of the program.

lexer.py

The module contains public and protected functions related to the lexical analysis stage. Actually, the only public function is *scan()* that receives an opened file object and scans the source code. The result of the process is a *Program* object that contains the program internal form and the two symbol tables.

▼ Interface

```
def scan(file: IO) -> Program:
    """ Lexically analyses the source code in the file. It returns a Program object
    containing the symbol table for ids and literals and the program internal form.

    Args:
        file (IO): opened file object

    Returns: a Program object

    Raises:
        LexicalError:
            if unexpected character is found
            if unterminated string literal is found
            if invalid character in int literal is found
            if leading zeros in int literal are found
    """
```

fa.py

The module contains the *FiniteAutomata* class and within the class there is a *State* structure representing a FA state that has a name and a list of transitions. A transitions is a key-value pair where the key is the symbol that the state can consume and the value is the next state.

The *FiniteAutomata* class is used for detected int literals, identifiers and keywords.

▼ State class

The class contains the name of the state and a dictionary of transitions. A transition is a key-value pair where the key is the symbol that the state is able to consume and the value is the next state if that symbol is going to be consumed.

▼ FiniteAutomata class

The class contains the alphabet as a list of symbols (strings), the states as a dictionary, where the key is the name of the state and the value the corresponding *State* instance, the initial state as a *State* instance and, finally, the final states as a list of *State* instances. The constructor of the class requires as parameter a file where the FA is described in order to load its elements. The user can check if a sequence is accepted by the FA by calling the *check_acceptance()* class method that receives an instance of *ImmutableBuffer* and yields the characters from the buffer that are accepted by the FA. If an character is not part of the FA's alphabet or a transition is not found for the current state and symbol an *InvalidSymbol* / *InvalidSequence* error is raised. Also, if the sequence found until that moment or after all the character of the buffer are consumed is not accepted an *InvalidSequence* error is raised.

Format of FA input file in EBNF

The FA input file is written in TOML and has a fixed structure. Here the FA's alphabet, states, inital / final states and transitions are described.

```
# ALPHABET
under_score = "_"

digit = "0" | ... | "9"
```

```
uppercase_letter = "A" | "B" | ... | "Z"
lowercase_letter = "a" | "b" | ... | "z"
letter = uppercase_letter | lowercase_letter

# STARTING RULE
start = alphabet_decl initial_state_decl final_states_decl states_decl

# GENERAL RULES
alphabet_decl = "alphabet" "=" symbols_array "\n"
initial_state_decl = "initial_state" "=" state "\n"
finial_states_decl = "final_states" "=" states_array "\n"
states_decl = { state_decl }

state_decl = "[[states]]" "\n" state_name_decl "\n" state_transitions_decl "\n"
state_name_decl = "name" "=" state
state_transitions_decl = "transitions" "=" transitions_array

transitions_array = "[" { transition "," } "]"
transition = "{" "symbol" "=" symbol "," "next_state" "=" state "}"

states_array = "[" { state "," } "]"
state = '"' ( underscore | letter | digit ) { underscore | letter | digit } '"'

symbols_array = "[" { symbol "," } "]"
symbol = '"' ( underscore | letter | digit ) { underscore | letter | digit } '"'
```

Example of FA input file

```
alphabet = [ "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" ]
initial_state = "A"
final_states = [ "B", "C" ]

[[states]]
name = "A"
transitions = [
    { symbol = "0", next_state = "B" },
    { symbol = "1", next_state = "C" },
    { symbol = "2", next_state = "C" },
    { symbol = "3", next_state = "C" },
    { symbol = "4", next_state = "C" },
    { symbol = "5", next_state = "C" },
    { symbol = "6", next_state = "C" },
    { symbol = "7", next_state = "C" },
    { symbol = "8", next_state = "C" },
    { symbol = "9", next_state = "C" },
]

[[states]]
name = "B"
transitions = []

[[states]]
name = "C"
transitions = [
    { symbol = "0", next_state = "C" },
    { symbol = "1", next_state = "C" },
    { symbol = "2", next_state = "C" },
    { symbol = "3", next_state = "C" },
    { symbol = "4", next_state = "C" },
    { symbol = "5", next_state = "C" },
    { symbol = "6", next_state = "C" },
```

```
    { symbol = "7", next_state = "C" },
    { symbol = "8", next_state = "C" },
    { symbol = "9", next_state = "C" },
]
```