Mnemo

Language Reference Manual

Avery Fan	Kathryn Harris	Soyoon Park	Kristine Pham	Dallas Scott
mf3332	kth2135	sp4412	klp2157	ds4015
System Architect	System Architect	Language Guru	Manager	Tester

Table of Contents

1. Introduction
2. Lexical Conventions
    2.1. Tokens
    2.2. Identifiers
    2.3. Keywords
    2.4. Literals
    2.5. Operators
    2.6. Separators
    2.7. Special Characters
3. Types
    3.1. Primitive Types
    3.2. Non-Primitive Types
4. Objects
    4.1. Object Attributes
5. Operators
    5.1. Arithmetic
    5.2. Assignment
    5.3. Equivalence
    5.4. Logical
    5.5. Other unary operator expressions
6. Statements/Expressions
    6.1. Declarations
    6.2. Literal expressions

6.3. Array Expressions
6.4. Control Flow
6.5. Sectional Tags
6.6. Branch Options
7. Functions
7.1. Standalone
7.2. Type-Based
7.3. Object-Based
8. Export Formatting
9. Grammar
9.1. Scanner
9.2. Parser
10. References

1. Introduction

Mnemo is a Python-based language for building a tree-based dialogue system, primarily aiming at programming text adventure games. Mnemo utilizes components of Python to provide tree and class structures specific to building a dialogue ecosystem. Through practical syntax and built-in tools, the user can conveniently modify and add characteristics and other modular components to decision trees and game characters. Mnemo will also support a built-in implementation of the dialogue to a basic graphical interface. Rather than developing an entire system to display their dialogue story, the users can have Mnemo automatically produce an interactive prototype of their game based on the dialogue after compilation.

Although this language is created for mainly game developers, Mnemo can be extendable to programming real-life use cases such as self-service kiosks, automated customer service, and interactive educational materials.

2. Lexical Conventions

2.1: Tokens

Tokens consist of identifiers, literal constants, reserved keywords, operators, separators, and special characters. Tokens are separated with whitespace, be it an actual space or a tab or newline.

Whitespace

`/new`

A newline can be appended to any `text` type, with a limit of 3 per `text` given the space limitations of the dialogue display box

```
alice: “Where do you think you’re going?\new I need
you here to help me.”
```

`/tab`

A tab can be included in any text type to indent part of the text.

```
bob: “I can’t help you because:\new \tab My leg is
broken.\new \tab I’m afraid of the dark.”
```

`Space ‘ ‘`

A space is the standard delimiter for tokens.

Comments

Any text following a # is a comment and will be ignored

```
# This is a comment
```

A multiline comment is formed using ` (grave) as both an opening and closing symbol. Anything between the grave symbols is ignored.

```
` This is
   a multiline
       comment `
```

2.2: Identifiers

Identifiers follow the regular expression pattern

```
(a-z | A-Z)(a-z | A-Z | 0-9 | _)*
```

They must start with a letter and then can contain letters in any case, digits, or the underscore character. Identifiers are case sensitive and cannot be reserved keywords.

Permitted:

```
narrator

Narrator

bag_of_fruit

char1

ornateGown

ALICE_and_BOB
```

Invalid:

`_potion`	cannot begin with an underscore
`0_apple`	cannot begin with a digit
`1_apple`	
`5Narrator`	
`label`	reserved keyword
`Alice&Bob`	ampersand not permitted

2.3: Keywords

The following are reserved keywords used in the language, either referencing types, Objects, or constructor parameters. They may not be used outside of those contexts.

Keyword	Description
`Narrative`	Used to define a Narrative object
`Character`	Used to define a Character object
`Item`	Used to define an Item object
`bool`	Implicit typing
`int`	Implicit typing
`float`	Implicit typing
`text`	Implicit typing
`array`	Implicit typing
`is`	Comparator
`inventory`	Parameter assignment
`label`	Parameter assignment
`next`	Parameter assignment
`hp`	Parameter assignment
`level`	Parameter assignment
`name`	Parameter assignment

2.4: Literals

Literal constants are enumerated by type, which is implicit and will be determined based on the usage pattern. These literals include int, float, bool, label, and text. Individual character literals are treated as text literals.

Int constant

A decimal digit from 0–9, not beginning with 0: `(1-9)(0-9)*`

```
4
32
563
80
```

Float constant

A floating point number containing decimal digits and a decimal mark. Can begin with 0 if immediately followed by the decimal mark, otherwise not: `((0.)|(1-9))(0-9)*.(0-9)*`

```
30.
5.6
0.8132
```

```
0.005
5.9311
7.5
```

Bool constant

A bool constant is either `True` or `False`.

Label constant

A string of characters not enclosed within any quotation marks. Can only contain digits, letters, and underscore:

```
((0-9)* | (a-z)* | (A-Z)* | _*)*
```

```
begin_story
Desert
Forest
refusedOffer
```

Text constant

A string of characters enclosed within opening and closing quotation marks:

```
“( (0-9)* | (~ | ! | @ | # | $ | % | ^ | &| * | ( | ) | _
| + | ` | / | { | } | [ | ] | < | > | ; | ’ | ? | . | ,)*
| (a-z)* | (A-Z)* )*”
```

```
    “hello WORLD”
    “01%32831***??\”
    “Please select an option.”
    “......”
    “I have 28 * (2.5 / 0.3) currency.”
```

The \ character serves as an escape to include a double quotation mark within the text literal.

Two adjacent `text` elements (including on separate lines) that are not broken by an assignment operation will be automatically concatenated.

```
    “Hello WORLD” “Please select an option”
         Output: “hello WORLDPlease select an option”


    alice: “hello WORLD”
    narrator: “Please select an option”
         No concatenation
```

A text literal can also be concatenated with a label literal using the ? operator:

```
    “You are now traveling to the ”?Forest
         Output: “You are now traveling to the Forest”
```

2.5: Operators

There are several types of operators which are more fully detailed in Chapter 5. These are logical operators, assignment operators, arithmetic operators, equivalence operators, and unary operators.

Logical Operators

^ (and) <> (or)

Arithmetic Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
+=	Compound Addition
-=	Compound Subtraction
*=	Compound Multiplication
/=	Compound Division

Equivalence Operators

Operator	Description
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`is`	Equal to

Assignment Operators

Operator	Description
`=`	Attribute assignment
`:`	Dialogue text assignment
`?`	Concatenation

Unary Operators

`~` (not) `++` (postfix increment) `--` (postfix decrement)

2.6: Separators

The below separators set off one token from another.

Separator	Usage
( )	Used to separate a function name from its parameters.
:	Used as both an assignment operator and as a separator to set off a Character variable from a text element.
,	Used to separate parameters in functions.
.	Used to separate a variable name from its attributes.

2.7: Special Characters

There are two special characters in Mnemo: one a signifier and one that represents both a function call and an attribute signifier.

`!` The label character signifies the beginning of a label literal. It follows the `label=` assignment and precedes the actual label literal.

```
alice: “This place is incredible.”, label=!enter_zone
```

`@` The option character signifies a branch option within a Node/node block. This both delineates the option text from standard dialogue text and also appends the option to the `options` array within the Node object.

```
/node
character1: “Please choose an option”
  @“Return me to the previous screen.”, next=!go_back
  @“Let’s go ahead”, next=!go_ahead
```

The option character must be indented after the dialogue text to signify that the option is being appended to that specific Node. The above appends two options to the `options` array/attribute of the node which precedes the options.

3. Types

Mnemo only makes use of a few types, both primitives and non-primitives. Some of these types have functions associated with them. Types in Mnemo are not explicit - they are inferred from both the content and context when used, though can be made explicit through the usage of an identifier function.

3.1: Primitive Types

`int`

A standard integer which holds a non-decimal number, either positive or negative. The int type is used for storing Object attributes, primarily in the Character and Item objects.

```
hp = 5
level += 1
dur = 25
```

`text`

A string of text that can contain spaces and is enclosed in quotation marks. The text type is used for Object names and is the main type for dialogue text in Node objects.

```
narrator: "Please choose an option."

Character("Alice", level=1, ...)
```

`label`

A label is similar to the text type, though it has the limitation that it must be unique when created (not referenced), is prefixed with `!`, cannot contain any spaces or special characters other than an underscore, and is not denoted with quotation marks. Labels are used as anchor point identifiers in the Narrative tree structure for branch points. They tell the nodes where to go after a tree branch.

```
alice: "Where am I?", label=!new_zone

Narrative("Beach Scene", root=!new_zone, narr_label=!beach)
```

`bool`

A true/false signifier. The bool type is used with an attribute in the Item object but is primarily used with comparators.

`float`

A decimal type for holding fractional numbers. While most variables will hold int types, the float is included in the language for those who desire more precision.

3.2: Non-Primitive Types

`array`

An array is a collection of Objects. It has no fixed length and is used primarily to store Item objects in a character's inventory and options in a node. Arrays have several functions associated with them.

4. Objects

Mnemo makes use of four primary objects, each with a variety of attributes and functions: the Narrative object serves as a decision tree; the Character object is used for character creation and management; the Node object makes up the fundamental element of the dialogue stream and Narrative tree; and the Item object is stored in a Character's inventory.

Narrative Object: The container for Nodes and overall structure for the decision tree. Narratives can be nested and are used to separate and organize various threads of the game's story.

```
story1 = Narrative ( title, root, narr_label )
```

Character Object: The object used to define a character in the game. These are essential, as each dialogue node must have a character associated with it.

```
alice = Character ( name, level, hp, inventory )
```

Node Object: The node is the fundamental element of the dialogue stream and makes up the building block of the Narrative tree. Nodes can have labels, links, and optional branches.

```
/node
Character_var: dialogue_text ; label
  @option, next
```

Item Object: An item object is what will be stored in a Character's inventory. Items have permanent or one-time usage, quantity, durability, uniqueness, cost, usage, and name attributes.

```
Item ( name, usage=[p|o], num, unique, dur, cost, cons )
```

4.1: Object Attributes

Narrative Attributes:

The attributes of the Narrative object are primarily descriptive. The `title` is a descriptive name for the story branch that does not have to be unique. The `root` links to the first node in the Narrative tree. The `narr_label` is a unique identifier, similar to the label for nodes, that serves as an anchor for linking Narrative objects with other nodes into a cohesive whole.

Type	Name	Description
`text`	`title`	Descriptive title for the tree
`label`	`root`	A link to the root node of the tree
`label`	`narr_label`	A unique identifier for the Narrative object that can be used to link from other nodes

Character Attributes:

The attributes of the Character object are those that will be displayed on screen during the game. The `name` is what will be displayed preceding all dialogue text associated with the character in the nodes. The `level` and `hp` attributes are self-explanatory. The `inventory` is a list of Item objects with a fixed length.

Type	Name	Description
`text`	`name`	Character name (displayed preceding dialogue text)
`int`	`level`	The character's level
`int`	`hp`	The character's hit points
`array`	`inventory`	An array of Item objects that belong to the character
`float`	`currency`	The amount of currency the character possesses

Node Attributes:

The attributes of the Node are varied. The only mandatory attribute is the `dialogue`, which contains the text of the character that is speaking. The `label` is used only on those nodes which follow an option branch. The `options` attribute is only used for a branch node where the dialogue can lead to several different paths in the Narrative. This takes the form of an array `[opt1 | next1, opt2 | next2, ...]` where `opt` is the display text for the option and `next` is a link to the node label that continues the branch. Each node has an internal `id` to determine sequencing.

Type	Name	Description
`text`	`dialogue`	`A string of dialogue text`
`label`	`label`	`An optional label for anchoring branches`
`array`	`options`	`Optional branch options`
`int`	`id`	`Internal id of node used for links`

Item Attributes:

The Item object has a mixture of visible and non-visible attributes. The `name` signifies the object's name. The `usage` attribute takes an assignment of either `p` or `o` to determine whether it is a permanent item or a one-time use item. The `num` attribute displays how many of the item the character has in his/her inventory. The `unique` attribute signifies whether only one of the item can be possessed at a time. The `dur` attribute is optional and is a numerical quantifier for the item's durability.

Type	Name	Description
`text`	`name`	`The name of the item`
`label`	`usage`	`A label determining either permanent (p) or one-time (o) usage; usage=p or usage=o`
`int`	`num`	`Quantity of the item in the user's inventory`
`bool`	`unique`	`A signifier for whether the item can be possessed in multiples or not`
`int`	`dur`	`The durability of the item; optional`
`float`	`cost`	`The cost of the item`
`bool`	`cons`	`Determines whether the item can be used (i.e., consumable)`

5. Operators

The operators in Mnemo are similar to those found in other languages, with a few minor exceptions. They are enumerated here with usage examples. Arithmetic operators are reserved for calculations to Character attributes. Most will be used in code sections rather than within nodes, with the exception of the assignment operator.

5.1: Arithmetic Operators

Addition: +

```
alice.hp = alice.hp + 50
```

Subtraction: -

```
if attack.hit: alice.hp = alice.hp - 250
```

Multiplication: `*`

```
alice.currency - potion.cost * 10
```

Division: `/`

```
alice.hp = alice.hp / attack.modifier
```

Compound Increment: `+=`

```
alice.level += 1
```

Compound Decrement: `-=`

```
alice.hp -= 100
```

Compound Multiplier: `*=`

```
alice.currency *= 10
```

Compound Divider: `/=`

```
alice.hp /= attack.modifier
```

5.2: Assignment Operators

There are three variants of the assignment operator. One is the standard = operator, another is used within nodes to tie a Character variable to a dialogue element, and the third concatenates a `label` type to a `text` type.

Equals: =

```
label=!branch_name
```

Colon :

```
alice: "Where am I?"
```

Concatenate ?

The concatenate operator concatenates a `label` type to a `text` type and assigns to a `text` type.

```
alice: "Stop that, please."?push_away
  Output:  alice: "Stop that, please.push_away"
```

5.3: Equivalence Operators

Less than: `<`

```
alice.hp < 500
```

Greater than: `>`

```
alice.level > 10
```

Greater than or equal: `>=`

```
if item.cost >= 100 then print("That one is expensive.")
```

Less than or equal: `<=`

```
if item.cost <= 20 then print("Take that one instead.")
```

Equal: `is`

```
if node.text is "" then print("error")
```

5.4: Logical Operators

and: `^`

```
if alice.level is 5 ^ alice.currency > 50
```

or: `<>`

```
potion.buy(5) <> food.buy(10)
```

5.5: Other Unary Operator Expressions

not: ~

```
if ~alice.inventory[tunic.dur] then print("Item is broken")
```

Postfix Increment: ++

```
alice.level++
```

Postfix Decrement: --

```
alice.inventory[potion.num]--
```

6. Statements/Expressions

6.1: Declarations

Code declaration

Within a code block delineated by a /code tag (see Chapter 6.5), a variable can be declared to an Object or a literal constant using the assignment operator =.

```
/code
player = Character(name="Player", level=1, hp=250, ...)
food.cost = 5
player_alive = True
greeting_msg = “Hello, welcome to the desert.”
```

Label declaration

Within a Node object or node block (see Chapter 6.5), a label can be declared to a Character variable and its dialogue text. This is so the user can jump back to this point in the dialogue stream if needed. The following format is shown below:

```
char1: “....” , label=<label>
```

```
    /node
    alice = “There seems to be a dark cave ahead. I will go check
    it out.”, label=enter_cave
    narrator = “Be careful on your way in. You might find what
    you are looking for...”
```

Label attributes are not required for a node. They can be included only if the node serves as a branch point for earlier options or is the root node in a Narrative tree. If there is no label, the second example above will be used (simply omit the `label=<label>` statement).

6.2: Literal Expressions

Literal expressions are fixed values written directly into the code. These values represent themselves and do not require evaluation or computation. Our language supports several types of literal expressions, similar to Python.

```python
42           # Integer literal expression
"hello"      # Text literal expression
start_game   # Label literal expression
True         # Boolean literal expression
None         # Null literal expression
[1, 2, 3]    # Array literal expression
```

6.3: Array Expressions

Arrays cannot be declared. They are internal attributes of Objects that can be used, but user-created arrays serve no function in the language and therefore cannot be created. The existing arrays are given by

```
Character.inventory = <array> Item[]
Node.options = <array> options[]
```

The array takes the form of either a sequence of Items delimited with commas or a sequence of options delimited with commas where each option has a secondary component in its array index, the `next` link. The `next` link is delimited from the option in this pairing with the `|` symbol.

```
[a, b, c]
[o1 | n1, o2 | n2]
```

Although new arrays cannot be declared, the two existing array attributes can be assigned to variables.

```
bag = alice.inventory
your_options = node.options
     result: bag and your_options are now arrays
```

Appending to an array: `<array>.add(), @`

```
@"Trust him", next=process_branch(options)
    appends an option to node.options with a node id as next
    [ ... , "Run away" | 1, "Trust him" | 3 ]
```

```
alice.inventory.add(apple)
    appends an Item object (apple) to alice's inventory
    [ ..., apple ]
```

Removing from an array: `<array>.remove()`

```
alice.inventory.remove(gown)
    removes the gown Item from alice's inventory
    [ potion, gown, apple ] -> [ potion, apple ]
```

6.4: Control Flow

Branch options within a node block introduces control flow to the overall dialogue stream. Additionally, a user can also utilize the keywords – `if, elif, else` – within a code block to further support back-end programming for the narrative.

```
/code
alice.level++
if alice.level > 10:
    alice.currency += 20
elif alice.level > 5:
    alice.currency += 10
else:
    alice.currency += 5
```

6.5: Sectional Tags

Mnemo is separated into code blocks and dialogue blocks. Sectional tags are differentiators that specify which section one is currently in. A dialogue block is also known as a node block. It is where dialogue text and option branches are enumerated. All other operations, including Object creation, attribute manipulation, function calls, arithmetic operations, etc. are performed inside code blocks.

```
/node
...
```

The node tag sets off a new dialogue element. It essentially serves as the constructor for the Node object in the form of a Character variable, dialogue text, optional option elements, and optional labels.

```
/nodes
...
```

The plural tag for defining multiple Nodes in one block (one per line).

```
/code
...
```

All other code is delineated with this tag

6.6: Branch Options

Branch options in our language provide the player with multiple choices that influence the flow of the narrative. These options are defined using the @ symbol, followed by the option text and the next attribute, which specifies the label of the next dialogue node that the game should transition to upon selection.

Upon seeing these options, the system would wait for the player's input.

```python
@option_text1, next=!label1
@option_text2, next=!label2

char1: bla, label=!label1
char2: blabla, label=!label2
```

`option_text`: The text displayed to the player as a selectable option.

`next=label_name`: Specifies the label of the next dialogue node that will execute if the player selects this option.

`process_options(options)`: This function prompts the player for input and returns the suitable next label upon return.

7. Functions

Mnemo functions come in three varieties: Standalone functions which are detached from any data structure, type-based functions which are utility functions for data types, and object-based functions that affect only the objects they are associated with.

7.1: Standalone Functions

`print(text)` return type: `None`

Prints `text` outside of the context of a dialogue box. The `print()` function is meant to be used for signifiers or game status changes and will primarily be used within `/code` sections outside of the context of a dialogue stream.

```
print("You purchased 4 potions.")
```

7.2: Type-Based Functions

Array:

`length()` return type: `int`

Returns the length of a given array as an integer.

```
alice.inventory.length()
     Ret: 4
```

`add(Item)` return type: None

Appends an item to the end of a given array.

```
alice.inventory.add(potion)
```

`remove(Item)` return type: None

Removes an item from the array.

```
alice.inventory.remove(tunic)
```

7.3: Object-Based Functions

Node Functions

`process_branch(options)` return: `node_id`

A node with branch `options` calls this function to request input from the player, select one of the options, and returns the `node_id` associated with that option. The `node_id` is then used to set the node's `next` link. Takes an array of options as a parameter.

Character Functions

`clear_inventory()` return: `None`

Delete all items from the character's inventory array.

```
alice.clear_inventory()
```

`set_name(text)` return: `None`

Sets or changes the name of a given character. Note that this does not change the variable name associated with the character. It only changes the `name` attribute and what will be displayed with the character's dialogue. For a complete change of variable name, a new Character object will need to be created.

```
alice.set_name("Bob")
      alice.name now returns "Bob"
```

Item Functions

`use()` return: `None`

Uses the item if the item has a `usage` attribute set. Reduces the `num` attribute of the item by 1.

```
potion.use()
      potion.num reduced by 1
```

`buy(quantity)` return: `None`

Increases the item's `num` attribute by `quantity` and reduces the Character's currency by `cost * quantity`.

```
food.buy(1)
      food.num increased by 1; Character.currency
      reduced by food.cost
```

`sell(quantity)` return: `None`

Reduces the item's `num` attribute by `quantity`, removes the item from the Character's inventory if `num` reaches 0, and increases the Character's currency by `cost * quantity`.

```
food.sell(5)
      food.num decreased by 5; Character.currency
      increased by 5 * food.cost
```

8. Export Formatting

Mnemo has an Export feature for easy sharing of game decision trees. It will export all Objects with their attributes and upon import will rebuild the code for the given game tree. Below is the formatting of the export file with one entry per Object type as a demonstration.

```
/Narratives
Narrative n1
      |_ text: story
      |_ label: begin_story
      |_ label: opening_scene
/Nodes
Node
      |_ text: The story begins in a remote mountainous region.
      |_ label: begin_story
      |_ array: [ “Climb rock” | 4, “Shout and hear echo” | 2 ]
      |_ int: 1
/Characters
Character alice
      |_ text: Alice
      |_ int: 10
      |_ int: 500
      |_ array: [ gown | bag_of_fruit ]
/Items
Item gown
      |_ text: Ornate Gown
      |_ label: p
      |_ int: 1
      |_ bool: true
      |_ int: 10
      |_ float: 30.
      |_ bool: false
```

9. Grammar

9.1: Scanner

```
{
open Buffer

let buffer = Buffer.create 256

let bool_of_string s =
  match String.lowercase_ascii s with
  | "true" -> true
  | "false" -> false
  | "True" -> true
  | "False" -> false
  | _ -> failwith "Invalid boolean string"

type token =
  | PLUS | MINUS | TIMES | DIVIDE            (* arithmetic operators *)
  | PLUSEQ | MINUSEQ | MULTEQ | DIVEQ
  | PLUSPLUS | MINUSMINUS                    (* unary operators *)
  | NOT | OR | AND | TRUE | FALSE            (* boolean operators*)
  | COLON | DOT | QUESTMARK | COMMA          (* separators *)
  | LPAREN | RPAREN | LBRACKET | RBRACKET
  | EQ | IS | LT | GT | LTE | GTE            (* equivalence operators *)
  | IF | ELSE | ELIF | RETURN                (* conditional expressions*)
  | NARRATIVE | CHARACTER | ITEM | NODE      (* keywords *)
  | NEXT | LABEL | HP | LEVEL
  | IDENT of string                          (* literals *)
  | INT_LITERAL of int
  | FLOAT_LITERAL of float
  | TEXT_LITERAL of string
  | LABEL_LITERAL of string
  | BOOL_LITERAL of bool
  | FUNCTION
  | AT
```

```
  | EOF
}
  rule token =
      (* whitespace *)
      parse [' '] { token lexbuf }
      (* literals *)
      | ['1'-'9']['0'-'9']* as i { INT_LITERAL (int_of_string i)}
      | ['0'-'9']+ ['.'] ['0'-'9']+ as f { FLOAT_LITERAL (float_of_string f) }
      | ['!'](['a'-'z' 'A'-'Z'] | ['0'-'9'] | '_')* as id { LABEL_LITERAL (id) }
      | "true" as b { BOOL_LITERAL (bool_of_string b) }
      | "false" as b { BOOL_LITERAL (bool_of_string b) }
      | "True" as b { BOOL_LITERAL (bool_of_string b) }
      | "False" as b { BOOL_LITERAL (bool_of_string b) }
      | "+" { PLUS }                    (* arithmetic operators *)
      | "-" { MINUS }
      | "*" { TIMES }
      | "/" { DIVIDE }
      | "+=" { PLUSEQ }
      | "-=" { MINUSEQ }
      | "*=" { MULTEQ }
      | "/=" { DIVEQ }
      | "." { DOT }
      | "Character" { CHARACTER }       (* keywords *)
      | "Item" { ITEM }
      | "/node" { NODE }
      | "next" { NEXT }
      | "label" { LABEL }
      | "=" { EQ }
      | "hp" { HP }
      | "level" { LEVEL }
      | "@" { AT }                      (* separators *)
      | ":" { COLON }
      | "?" { QUESTMARK }
      | "," { COMMA }
      | "(" { LPAREN }
      | ")" { RPAREN }
      | "[" { LBRACKET }
      | "]" { RBRACKET }
      | "return" { RETURN }
      | "if" { IF }                     (* conditionals *)
```

```
      | "else" { ELSE }
      | "elif" { ELIF }
      | "~" { NOT }                    (* boolean operators *)
      | "^" { AND }
      | "<>" { OR }
      | "++" { PLUSPLUS }              (* unary operators *)
      | "--" { MINUSMINUS }
      | "is" { IS }                    (* equivalence operators *)
      | "<" { LT }
      | ">" { GT }
      | ">=" { GTE }
      | "<=" { LTE }
      | "#" [^'\n']* { token lexbuf }  (* comment one line *)
      | "`" { comment_ml lexbuf }      (* comment multiline *)
      | '"' { scan_text lexbuf }       (* ignore in text *)
      | ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z'] | ['0'-'9'] | '_')* as id { IDENT (id) }
      | eof { EOF }

(* inside comment *)
and comment_ml = parse
      "`" { token lexbuf }
      | _ { comment_ml lexbuf }

(* inside string *)
and scan_text = parse
  | '"' { TEXT_LITERAL (Buffer.contents buffer) }
  | "\\tab" { Buffer.add_char buffer '\t'; scan_text lexbuf }
  | "\\new" { Buffer.add_char buffer '\n'; scan_text lexbuf }
  | [^'"']+ as str {
      Buffer.add_string buffer str;
      scan_text lexbuf
  }
  | eof { failwith "String not closed" }
```

9.2: Parser

```
%{
  open Scanner
%}

type operator = Add | Sub | Mul | Div | Eq | Geq | Leq | Lt | Gt
type unaryop = Incr | Decr

let apply_op op left right =
  match op with
  | Add -> left + right
  | Sub -> left - right
  | Mul -> left * right
  | Div -> left / right
  | Eq -> left = right
  | Geq -> left >= right
  | Leq -> left <= right
  | Lt -> left < right
  | Gt -> left > right
  | Incr -> left + 1
  | Decr -> left - 1
  | _ -> failwith "Unsupported operator"

type expr =
    Binop of expr * operator * expr
  | Unop of expr * unaryop
  | IntLit of int
  | TextLit of string
  | BoolLit of bool
  | LabelLit of string
  | FloatLit of float
  | Ident of string
  | Assign of string * expr

let rec eval expr =
  match expr with
  | IntLit n -> n
  | TextLit n -> n
```

```
  | BoolLit n -> n
  | LabelLit n -> n
  | FloatLit n -> n
  | Ident n -> n
  | Binop (left, op, right) ->
      let left_val = eval left in
      let right_val = eval right in
      apply_op op left_val right_val
  | Unop (e, op) ->
      let val_e = eval e in
      apply_op op val_e
  | Assign (left, right) ->
        let left_val = eval left in
        let right_val = eval right in
        apply_op op left_val right_val
  | _ -> failwith "Unknown expression"

type stmt =
  | If of expr * stmt * stmt
  | Expr of expr

type node = {
  character: string;
  id: int;
  dialogue: string;
  label: string;
  options: (string * expr) list;
  next: string
}

type item = {
  name: string;
  usage: string;
  unique: bool;
  cost: float;
  dur: int;
  num: int;
  unique: bool;
}
```

```
type chrctr = {
      name: string;
      level: int;
      hp: int;
      inventory: item list;
}

type narr = {
      title: string;
      root: string;
      narr_label: string;
}

%token PLUS MINUS TIMES DIVIDE
%token PLUSEQ MINUSEQ MULTEQ DIVEQ
%token PLUSPLUS MINUSMINUS
%token NOT OR AND
%token COLON DOT COMMA QUESTMARK
%token LPAREN RPAREN LBRACKET RBRACKET
%token EQ IS LT GT LTE GTE
%token IF ELSE ELIF RETURN
%token NARRATIVE CHARACTER ITEM NODE
%token NEXT LABEL HP LEVEL
%token AT EXCLMPT
%token <int> INT_LITERAL
%token <bool> BOOL_LITERAL
%token <float> FLOAT_LITERAL
%token <string> LABEL_LITERAL
%token <string> TEXT_LITERAL
%token <string> IDENT
%token EOF EOL

%right ASSIGN
%left OR
%left AND
%left IS NOT GT LT GTE LTE
%left PLUS MINUS
%left TIMES DIVIDE
%nonassoc UMINUS
```

```
program_rule:
  vdecl_list_rule stmt_list_rule EOF { {locals=$1; body=$2} }

vdecl_list_rule:
  /*nothing*/ { [] }
  | vdecl_rule_narr vdecl_rule_char vdecl_rule_item vdecl_rule_node
    vdecl_list_rule { $1 :: ($2 :: $3 :: $4 :: $5)  }

vdecl_rule_narr:
  IDENT EQ NARRATIVE LPAREN TEXT_LITERAL COMMA LABEL_LITERAL COMMA
  LABEL_LITERAL RPAREN { title: $5, root: $7, narr_label: $9 }

vdecl_rule_char:
  IDENT EQ CHARACTER LPAREN TEXT_LITERAL COMMA INT_LITERAL COMMA
  INT_LITERAL COMMA IDENT RPAREN { name: $5, level: $7, hp: $9, inventory:
  [], currency: $13 }

vdecl_rule_item:
  IDENT EQ ITEM LPAREN IDENT COMMA IDENT COMMA INT_LITERAL COMMA
  BOOL_LITERAL COMMA INT_LITERAL COMMA INT_LITERAL COMMA BOOL_LITERAL
  RPAREN { name: $5, usage: $7, num: $9, unique $11, dur: $13, cost: $15,
  cons: $17 }

vdecl_rule_node:
  (* node without label, with options *)
  NODE IDENT COLON TEXT_LITERAL opt_list {
      let node_id = !node_counter in
    node_counter := !node_counter + 1;
    { id = next_node_id(); dialogue = $4; label = ""; options = $5 }
  }
  (* node without label or options *)
  | NODE IDENT COLON TEXT_LITERAL
  {
    let node_id = !node_counter in
    node_counter := !node_counter + 1;
    { id = node_id; dialogue = $4; label = ""; options = [] }
  }
  (* node with label, without options)
  | NODE IDENT COLON TEXT_LITERAL COMMA LABEL EQ LABEL_LITERAL
  {
```

```
    let node_id = !node_counter in
    node_counter := !node_counter + 1;
    { id = node_id; dialogue = $4; label = $8; options = [] }
  }
  (* node with label and options *)
  | NODE IDENT COLON TEXT_LITERAL COMMA LABEL EQ LABEL_LITERAL opt_list
  {
    let node_id = !node_counter in
    node_counter := !node_counter + 1;
    { id = node_id; dialogue = $4; label = $8; options = $9 }
  }

(* node options *)
type option_expr =
  | TextLit of string

opt_list:
  /* Empty */ { [] }
  | opt_list option_expr { $2 :: $1 }

option_expr:
  | AT TEXT_LITERAL COMMA NEXT EQ IDENT
      { $2 }

stmt_list_rule:
    /* nothing */               { []     }
    | stmt_rule stmt_list_rule  { $1::$2 }

stmt_rule:
  expr_rule { Expr $1 }
  | IF expr_rule COLON stmt_rule ELSE COLON stmt_rule
      { If ($2, $4, $7) }

expr_rule:
  INT_LITERAL { IntLit $1 }
  | BOOL_LITERAL { BoolLit $1 }
  | TEXT_LITERAL { TextLit $1 }
  | LABEL_LITERAL { LabelLit $1 }
  | FLOAT_LITERAL { FloatLit $1 }
  | IDENT { Ident $1 }
```

```
| LPAREN expr_rule RPAREN { $2 }
| expr_rule PLUS expr_rule { Binop ($1, Add, $3) }
| expr_rule MINUS expr_rule { Binop ($1, Sub, $3) }
| expr_rule TIMES expr_rule { Binop ($1, Mul, $3) }
| expr_rule DIVIDE expr_rule { Binop ($1, Div, $3) }
| expr_rule GT expr_rule { Binop ($1, Gt, $3) }
| expr_rule LT expr_rule { Binop ($1, Lt, $3) }
| expr_rule LTE expr_rule { Binop ($1, Leq, $3) }
| expr_rule GTE expr_rule { Binop ($1, Geq, $3) }
| expr_rule PLUSPLUS { Unop ($1, Incr) }
| expr_rule MINUSMINUS { Unop ($1, Decr) }
| NOT expr_rule { Unop ($2, Not) }
| IDENT EQ expr_rule { Assign ($1, $3) }
```

10. References

- Rusty LRM: https://www.cs.columbia.edu/~sedwards/classes/2016/4115-fall/lrms/rusty.pdf
- Python LRM: https://docs.python.org/3/reference/index.html
- The GNU C Reference Manual: https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html