DATA C88C
Spring 2026

Abstract Data Types, Recursion

Discussion 5: February 23, 2026

Abstract Data Types

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects — for example, car objects, chair objects, people objects, etc. That way, programmers don't have to worry about *how* code is implemented — they just have to know *what* it does.

Data abstraction mimics how we think about the world. For example, when you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of. You just have to know how to turn the wheel and press the gas pedal.

An **abstract data type** consists of two types of functions:

- **Constructors**: functions that build the abstract data type.
- **Selectors**: functions that retrieve information from the data type.

Q1: Word

In this problem, we will implement two data abstractions from scratch to represent a language! Let's first build an abstraction for words, which will compose each language.

The `word` abstraction stores: 1. a word's `name` 2. a word's `definition` in English

Your first job will be to create the *constructors* and *selectors* for the `word` type *in two ways* (denoted by `a` and `b`).

- A clean, typical approach would be to use a `list` pair to bundle together attributes of our type. Can you come up with two other implementations of the word ADT? Some ideas include using a dictionary or higher-order functions!

  Note: Concerning ourselves with the implementation of this ADT does put us *"beneath" the abstraction barrier*. However, a takeaway from this problem is that as we move towards higher-level functionalities (like translation), we no longer have to worry about the specifics of our original implementation. These low-level details are *abstracted away* by our constructors and selectors!

```
#Implementation a
def make_word_a(name, definition):
    """
    >>> yes = make_word_a('yes', 'affirmative response')
    >>> get_word_name_a(yes), get_word_definition_a(yes)
    ('yes', 'affirmative response')
    """
    "*** YOUR CODE HERE ***"




def get_word_name_a(word):
    "*** YOUR CODE HERE ***"




def get_word_definition_a(word):
    "*** YOUR CODE HERE ***"





#Implementation b
def make_word_b(name, definition):
    """
    >>> yes = make_word_b('yes', 'affirmative response')
    >>> get_word_name_b(yes), get_word_definition_b(yes)
    ('yes', 'affirmative response')
    """
    "*** YOUR CODE HERE ***"




def get_word_name_b(word):
    "*** YOUR CODE HERE ***"




def get_word_definition_b(word):
    "*** YOUR CODE HERE ***"





def check_word_abstraction():
    """Check whether both constructors and selectors work properly.
    >>> yes_a = make_word_a('yes', 'affirmative response')
    >>> yes_b = make_word_b('yes', 'affirmative response')
    >>> get_word_name_a(yes_a) == get_word_name_b(yes_b)
    True
    >>> get_word_definition_a(yes_a) == get_word_definition_b(yes_b)
    True
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Q2: Language

Now, implement an abstraction for language. The `language` ADT stores:

1. a language's `name`
2. a list of `word`'s present in the language

Finally, using both the `word` and `language` ADTs, implement the two following functions:

- `translate`, which takes `source_word`, a `word` abstraction, and returns the `name` of the translated word in the `target` language. If the word cannot be found, return `'Undefined'`.
- `update`, which updates a language's list of words with `new_word`.

  Hint: When we translate a word to a different language, what attribute of the word remains the same? From there, it's just about searching for and accessing that attribute at the right location!

  Hint: With abstract data types, we can construct ("create") and select ("read") data, but we can't exactly mutate them. To "update" a language, we should instead construct and return a new language altogether!

```
# Paste your word ADT here!
def make_word(name, definition):
    """
    >>> yes = make_word('yes', 'affirmative response')
    >>> get_word_name(yes), get_word_definition(yes)
    ('yes', 'affirmative response')
    """
    "*** YOUR CODE HERE ***"




def get_word_name(word):
    "*** YOUR CODE HERE ***"




def get_word_definition(word):
    "*** YOUR CODE HERE ***"




# Implement the language ADT here!
def make_language(name, words):
    """
    >>> hi, there = make_word('hi', 'friendly greeting'), make_word('there', 'at that
 place')
    >>> english = make_language('english', [hi, there])
    >>> get_language_name(english)
    'english'
    >>> for w in get_language_words(english):
    ...     print(get_word_name(w))
    ...
    hi
    there
    """
    "*** YOUR CODE HERE ***"




def get_language_name(language):
    "*** YOUR CODE HERE ***"




def get_language_words(language):
    "*** YOUR CODE HERE ***"




def translate(source_word, target):
    """Given a word, translate it into the 'target' language.
```

Recursion

Recursion is when a function calls itself to solve a smaller version of the same problem. Instead of tackling a complex task all at once, recursion breaks it down into simpler steps until it reaches a **base case**.

Many students find this topic challenging. Everything gets easier with practice. Please help each other learn.

Q3: Swipe

Implement `swipe`, which prints the digits of argument `n`, one per line, **first backward then forward**. The left-most digit is printed only once. **Do not use** `while` or `for` or `str`. (Use recursion, of course!)

```
def swipe(n):
    """Print the digits of n, one per line, first backward then forward.

    >>> swipe(2837)
    7
    3
    8
    2
    8
    3
    7
    """
    if n < 10:
        print(n)
    else:
        "*** YOUR CODE HERE ***"
```

Q4: Skip Factorial

Define the base case for the `skip_factorial` function, which returns the product of **every other positive** integer, starting with `n`.

```
def skip_factorial(n):
    """Return the product of positive integers n * (n - 2) * (n - 4) * ...

    >>> skip_factorial(5) # 5 * 3 * 1
    15
    >>> skip_factorial(8) # 8 * 6 * 4 * 2
    384
    """
    if ___:
        return ___
    else:
        return ___
```

Q5: Recursive Hailstone

Recall the `hailstone` function from Homework 2. First, pick a positive integer `n` as the start. If `n` is even, divide it by 2. If `n` is odd, multiply it by 3 and add 1. Repeat this process until `n` is 1. Complete this recursive version of `hailstone` that prints out the values of the sequence and returns the number of steps.

```
def hailstone(n):
    """Print out the hailstone sequence starting at n,
    and return the number of elements in the sequence.
    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    >>> b = hailstone(1)
    1
    >>> b
    1
    """
    print(n)
    if n % 2 == 0:
        return even(n)
    else:
        return odd(n)

def even(n):
    return ____

def odd(n):
    "*** YOUR CODE HERE ***"
```