Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Υλοποίηση Βιβλιοθήκης Ημερομηνίας και Ώρας για την Γλώσσα Προγραμματισμού Rust

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΛΦΡΕΔΟΣ-ΠΑΝΑΓΙΩΤΗΣ Λ. ΔΑΜΚΑΛΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2015

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών

Υλοποίηση Βιβλιοθήκης Ημερομηνίας και Ώρας για την Γλώσσα Προγραμματισμού Rust

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΛΦΡΕΔΟΣ-ΠΑΝΑΓΙΩΤΗΣ Λ. ΔΑΜΚΑΛΗΣ

Επιβλέπων : Νικόλαος Σ. Παπασπύρου
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 23η Οκτωβρίου 2015.

.....	.....	.....
Νικόλαος Σ. Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.	Κωνσταντίνος Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Κωνσταντίνος Κοντογιάννης Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2015

Αλφρέδος-Παναγιώτης Λ. Δαμκαλής

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Σκοπός της παρούσας εργασίας ήταν η δημιουργία μιας βιβλιοθήκης ημερομηνίας και ώρας για την γλώσσα προγραμματισμού Rust, η οποία μοντελοποιεί αναλυτικά την έννοια του χρόνου υλοποιώντας τα επί μέρους κομμάτια που τον συνθέτουν και επιτρέποντας τις μεταξύ τους πράξεις καθώς και τον εύκολο χειρισμό τους.

Η έννοια του χρόνου όπως την αντιλαμβανόμαστε καθημερινά φαίνεται απλή, ωστόσο όταν αρχίζουμε να την εξετάζουμε πιο διεξοδικά γίνεται όλο και πιο πολύπλοκη. Η ύπαρξη πολλών εννοιών, πολλές φορές με πολλαπλό νόημα, προερχόμενες από το μακρινό έως το πιο πρόσφατο παρελθόν κάνουν τα πράγματα πιο σύνθετα.

Κατά την σχεδίαση της βιβλιοθήκης έπρεπε να ληφθούν υπόψιν όλες αυτές οι παράμετροι που επηρεάζουν τον χρόνο, έτσι όπως τον παρατηρούμε και τον μετράμε σήμερα και να λυθούν μια σειρά από ζητήματα που σχετίζονται με αυτές. Πολύτιμη βοήθεια σε αυτή την προσπάθεια υπήρξαν οι προδιαγραφές JSR-000310 από τις οποίες επηρεάστηκαν πολλές αποφάσεις κατά την υλοποίηση της βιβλιοθήκης.

Λέξεις κλειδιά

Γλώσσα προγραμματισμού Rust, Βιβλιοθήκη προγραμματισμού, Ημερομηνία, Ώρα, Χρόνος, πρότυπο ISO 8601.

Abstract

The purpose of this diploma dissertation was the implementation of a programming library of date and time for Rust programming language. This library offers a comprehensive model for date and time and allows operations between date-time objects and an easy way to manipulate them. Date and Time look like something simple from our daily experience, however when start to look at them more carefully they get more and more complicated. Many concepts around date and time, some of them with different meanings, coming from the past add more complexity. Parameters that affect date and time, as we observe and measure them today, should be taken into consideration during designing the library. Furthermore, some emerged issues related with these parameters should be solved. In this situation JSR-000310 specifications offer a valuable help and had played a significant role on many decisions taken during building this library.

Key words

Rust programming language, Programming Library, Date, Time, ISO 8601.

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα αυτής της διπλωματικής εργασίας, κ. Νίκο Παπασπύρου, για τον χρόνο που μου αφιέρωσε, την υποστήριξή του και την υπομονή του καθόλη την διάρκεια εκπόνησης της εργασίας μου. Επίσης θέλω να ευχαριστήσω την οικογένειά μου που στάθηκε δίπλα μου όλα αυτά τα χρόνια των σπουδών μου. Τέλος, ένα ευχαριστό σε όλους τους καθηγητές και τους συμφοιτητές μου που συμπορευτήκαμε μαζί κατά την ακαδημαϊκή μου διαδρομή.

Αλφρέδος-Παναγιώτης Λ. Δαμκαλής,

Αθήνα, 23η Οκτωβρίου 2015

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-6-15, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Οκτώβριος 2015.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Σκοπός

Σκοπός της παρούσας διπλωματικής εργασίας ήταν η υλοποίηση βιβλιοθήκης προγραμματισμού για τον χειρισμό του χρόνου (ημερομηνίας και ώρας) για την γλώσσα προγραμματισμού Rust, βασισμένη στην διεπαφή εφαρμογών προγραμματισμού (API) για την ημερομηνία και ώρα της πλατφόρμας JDK 8 που υλοποιεί τις προδιαγραφές JSR-000310.

1.2 Η Γλώσσα Προγραμματισμού Rust

Η γλώσσα Rust είναι μία γλώσσα προγραμματισμού συστημάτων που εστιάζει στην ασφάλεια, στην ταχύτητα και στον ταυτοχρονισμό (concurrency). Για να πετύχει αυτούς τους στόχους χρησιμοποιεί ένα σύνολο ιδεών, με πολλές από αυτές να προέρχονται από άλλες γλώσσες προγραμματισμού.

Ο κύριος σκοπός της γλώσσας είναι η εξασφάλιση της ασφάλειας της μνήμης (memory safety), η εξάλειψη συνθηκών ανταγωνισμού δεδομένων (data races) και η μηδενικού κόστους αφαίρεση (zero-cost abstraction) έτσι ώστε να παράγει προγράμματα με ασφαλή και γρήγορη εκτέλεση.

Τα παραπάνω σε συνδυασμό με το στατικό και δυνατό (strong) σύστημα τύπων της, την ενσωμάτωση σε αυτό του συστήματος ιδιοκτησίας/δανεισμού (ownership/borrowing system), την έλλειψη συλλέκτη απορριφθέντων στοιχείων (garbage collector), την ευκολία ενσωμάτωσης κώδικα από άλλες γλώσσες ή σε άλλες γλώσσες, καθώς και τον έλεγχο σε χαμηλό επίπεδο (low-level control) κάνουν την γλώσσα Rust μία καλή εναλλακτική πρόταση ασφαλούς γλώσσας προγραμματισμού, συγκρίσιμη σε απόδοση με άλλες γλώσσες όπως η C ή η C++.

1.3 Χρόνος - Ημερομηνία και Ώρα

Η έννοια του χρόνου είναι άρρηκτα συνδεδεμένη με τον άνθρωπο και την καθημερινότητά του. Τα πρώτα καταγεγραμμένα δείγματα μέτρησής του έρχονται από την εποχή του χαλκού, ενώ πιστεύεται ότι διάφορες νεολιθικές κατασκευές που έχουν ανακαλυφθεί μπορεί να χρησίμευαν και αυτές για την μέτρηση του. Από τότε η τεχνολογία και οι γνώσεις του ανθρώπινου γένους έχουν εξελιχθεί και με αυτές έχει εξελιχθεί και ο τρόπος που ο άνθρωπος αντιλαμβάνεται, παρατηρεί, χρησιμοποιεί και μετράει τον χρόνο.

Κατά την διάρκεια του περασμένου αιώνα, το εμπόριο καθώς και η ανάπτυξη των μέσων μεταφοράς και των τηλεπικοινωνιών κατέστησαν αναγκαία την προτυποποίηση του χρόνου με την υιοθέτηση κοινού ημερολογίου, κοινής ώρας και κοινών προτύπων για την αναπαράστασή του. Ωστόσο παρότι υπάρχουν πλέον διεθνή πρότυπα αυτά δεν ακολουθούνται πάντα, είτε ακολουθούνται μερικώς εξαιτίας κοινωνικών, πολιτικών, γεωγραφικών ή θρησκευτικών λόγων.

1.4 Προδιαγραφές JSR-000310

Οι προδιαγραφές JSR-000310 σχεδιάστηκαν για να αποτελέσουν τις οδηγίες κατασκευής της νέας διεπαφής εφαρμογών προγραμματισμού (API) για την ημερομηνία και ώρα της πλατφόρμας JDK 8. Περιγράφουν αναλυτικά μοντέλα για την ημερομηνία και την ώρα και βασίζονται στο διεθνές πρότυπο ISO-8601. Ορίζουν αμετάβλητους τύπους τιμών (immutable value types) έτσι ώστε να είναι εύκολη και συμβατή η χρήση τους σε πολυπύρηνα περιβάλλοντα και προσφέρουν μία κατανοητή και πλούσια διεπαφή. Τέλος υποστηρίζουν κάποια τοπικά συστήματα ημερολογίων και δίνουν την δυνατότητα για υποστήριξη περισσοτέρων.

1.5 Δομή Εργασίας

Η εργασία ακολουθεί την παρακάτω δομή:

- **Κεφάλαιο 2:** Στο κεφάλαιο αυτό περιγράφονται τα βασικότερα στοιχεία της γλώσσας Rust τα οποία χρησιμοποιήθηκαν κατά την δημιουργία της βιβλιοθήκης.
- **Κεφάλαιο 3:** Σε αυτό το κεφάλαιο γίνεται η ανάλυση των εννοιών της ημερομηνίας και της ώρας καθώς και σχετικών ζητημάτων που τις ακολουθούν.
- **Κεφάλαιο 4:** Στο κεφάλαιο αυτό αναλύεται η δομή και η χρήση της βιβλιοθήκης καθώς και θέματα που προέκυψαν κατά την υλοποίησή της.
- **Κεφάλαιο 5:** Το κεφάλαιο αυτό περιέχει τα συμπεράσματα της εργασίας και κατευθύνσεις για μελλοντική έρευνα.

Κεφάλαιο 2

Περιγραφή της Γλώσσας Προγραμματισμού Rust

2.1 Ιστορική Αναδρομή

Η γλώσσα Rust ξεκίνησε το 2006 ως δευτερεύον πρότζεκτ του Γκρέιντον Χόαρ (Graydon Hoare) και αναπτύχθηκε ως τέτοιο για τα επόμενα τρία χρόνια. Το 2009, όπου η γλώσσα είχε πάρει μία πιο ώριμη μορφή και μπορούσε να υποστηρίξει τον σκοπό της, ενσωματώθηκε στα πρότζεκτ του Mozilla και επισήμως ανακοινώθηκε το 2010.

Από το Mozilla, η Rust, χρησιμοποιήθηκε και χρησιμοποιείται μέχρι και σήμερα κυρίως για την ανάπτυξη της πειραματικής μηχανής διάταξης (layout engine) Servo. Καθώς η μηχανή Servo χρησιμοποίησε την γλώσσα πολύ πριν από την σταθερή της έκδοση δοκίμασε, έλεγξε και επηρέασε αρκετά από τα χαρακτηριστικά της.

Παρότι το Mozilla χρηματοδοτεί την ανάπτυξη της γλώσσας, πολλές από τις συνεισφορές στην Rust γίνονται από την κοινότητά της, η οποία σήμερα αποτελείται από ανθρώπους με διαφορετικά πεδία ενδιαφέροντος.

Ο μεταγλωττιστής της γλώσσας Rust, γνωστός και ως rustc, γράφτηκε αρχικά στην γλώσσα OCaml, ενώ στην συνέχεια στην ίδια την γλώσσα Rust. Η πρώτη αριθμημένη πρωταρχική (pre-alpha) έκδοση του ήταν διαθέσιμη τον Ιανουάριο του 2012 και μετά από αρκετές αλλαγές στην γλώσσα και στον ίδιο τον μεταγλωττιστή έφτασε στην σταθερή του έκδοση 1.0 στις 15 Μαΐου 2015. Σήμερα βρίσκεται στην σταθερή έκδοση 1.3 και ακολουθεί το μοντέλο συνεχούς ενσωμάτωσης (continuous intergration) με καινούρια σταθερή έκδοση κάθε έξι εβδομάδες.

2.2 Συντακτικό και Σημασιολογία

Όπως αναφέρθηκε η γλώσσα Rust έχει ως στόχο την ασφαλή και γρήγορη εκτέλεση των προγραμμάτων που είναι γραμμένα σε αυτή και το επιτυγχάνει με διάφορους τρόπους. Στην ενότητα που ακολουθεί παρουσιάζονται κάποιοι από αυτούς.

2.2.1 Βασικοί τύποι

Μοναδιαίος τύπος (Unit type):

Ο μοναδιαίος τύπος εκφράζεται με άνοιγμα και κλείσιμο παρένθεσης `()`, και είναι το αποτέλεσμα μίας έκφρασης που δεν επιστρέφει κάποια τίμη.

Δυαδικός τύπος (Boolean):

Ο δυαδικός τύπος ονομάζεται `bool` και έχει δύο τιμές, `true` και `false`.

Τύπος χαρακτήρα (Char):

Ο τύπος χαρακτήρα ονομάζεται `char` και αντιπροσωπεύει έναν χαρακτήρα Unicode. Η δημιουργία του τύπου αυτού περιλαμβάνει τον χαρακτήρα εντός μονών εισαγωγικών, για παράδειγμα `'S'` ή `'é'`.

Αριθμητικοί τύποι (Numeric types):

Η Rust διαθέτει αρκετούς αριθμητικούς τύπους χωρισμένους σε κατηγορίες: προσημασμένοι ή χωρίς πρόσημο, σταθερού ή μεταβλητού μεγέθους, κινητής υποδιαστολής ή ακέραιοι:

Πίνακας 2.1: Αριθμητικοί τύποι της γλώσσας Rust.

Τύπος	Κατηγορία	Μέγεθος
`i8`	Προσημασμένος ακεραιος	8 bits
`i16`	Προσημασμένος ακεραιος	16 bits
`i32`	Προσημασμένος ακεραιος	32 bits
`i64`	Προσημασμένος ακεραιος	64 bits
`u8`	Ακέραιος χωρίς πρόσημο	8 bits
`u16`	Ακέραιος χωρίς πρόσημο	16 bits
`u32`	Ακέραιος χωρίς πρόσημο	32 bits
`u64`	Ακέραιος χωρίς πρόσημο	64 bits
`isize`	Προσημασμένος ακεραιος	μεταβλητό
`usize`	Ακέραιος χωρίς πρόσημο	μεταβλητό
`f32`	Κινητής υποδιαστολής	32 bits
`f64`	Κινητής υποδιαστολής	64 bits

Το μέγεθος των τύπων `isize` και `usize` εξαρτάται από την αρχιτεκτονική του συστήματος στο οποίο γίνεται η μεταγλώττιση του προγράμματος. Οι τύποι κινητής υποδιαστολής αντιστοιχούν στους αριθμούς απλής και διπλής ακρίβειας IEEE-754 [12].

Σε περίπτωση όπου δεν μπορεί να εξαχθεί ο αριθμητικός τύπος ενός αριθμού τότε χρησιμοποιείται ως προεπιλεγμένος τύπος ο `i32` και ο `f64` για τους ακέραιους αριθμούς και τους αριθμούς κινητής υποδιαστολής αντίστοιχα. Τέλος υπάρχει η δυνατότητα χρήσης του χαρακτήρα `_` όπου χρησιμεύει για διαχωρισμό και αγνοείται από τον μεταγλωττιστή για παράδειγμα `1_000_000`, `2_u8` ή `3.456_234f64`.

Πίνακες (Arrays):

Οι πίνακες είναι σταθερού μεγέθους και περιέχουν στοιχεία του ίδιου τύπου. Ως προεπιλογή οι πίνακες είναι αμετάβλητοι (immutable) και ορίζονται με τον παρακάτω τρόπο:

```
[T; N]
```

όπου `T` είναι ο τύπος των στοιχείων του πίνακα και `N` το μέγεθος του πίνακα που είναι γνωστό κατά την μεταγλώττιση, για παράδειγμα `[i64, 5]`. Για συντομία η αρχικοποίηση ενός πίνακα μπορεί να γίνει χρησιμοποιώντας την αρχική τιμή στην θέση όπου δίνουμε τον τύπο `T`, για παράδειγμα `[0u32; 20]`.

Η πρόσβαση στα στοιχεία του πίνακα γίνεται χρησιμοποιώντας την μεταβλητή που έχει δεθεί με τον πίνακα και την θέση του στοιχείου στον πίνακα εντός τετράγωνων αγκύλων με την πρώτη θέση να αντιστοιχεί στο `0`:

```
{
    let items: [u32; 5] = [1, 2, 3, 4 ,5];
    items[2]; // 3
}
```

Τομές (Slices):

Οι τομές είναι αναφορές σε άλλες δομές δεδομένων και είναι χρήσιμες γιατί επιτρέπουν την ασφαλή και εύκολη πρόσβαση σε μέρος μιας δομής δεδομένων χωρίς το κόστος της αντιγραφής.

Οι τομές έχουν μέγεθος και μπορεί να είναι μεταβλητές (mutable) ή αμετάβλητες και σε πολλές περιπτώσεις συμπεριφέρονται όπως οι πίνακες:

```
{
    let items: [u32; 5] = [1, 2, 3, 4 ,5];
    let slice = &items[1..3] // &[2, 3 ,4]
    slice[2]; // 4
}
```

Βασικός τύπος συμβολοσειράς (str):

Πρόκειται για μία σειρά Unicode τιμών κωδικοποιημένων ως ροή UTF-8 bytes. Επιπλέον δεν τερματίζονται με μηδενική τιμή (not null-terminated) και ως εκ τούτου μπορούν να φέρουν και μηδενικές τιμές. Ο τύπος αυτός δεν έχει ορισμένο μέγεθος και για αυτόν τον λόγο η πρόσβαση σε αυτόν γίνεται μέσω αναφορών, για παράδειγμα `&str`. Για να δηλώσουμε μία αναφορά σε συμβολοσειρά γράφουμε την συμβολοσειρά εντός διπλών εισαγωγικών:

```
{
    let string: &str = ”One String”;
}
```

Πλειάδες (Tuples):

Οι πλειάδες αποτελούν ταξινομημένες λίστες σταθερού μεγέθους όπου τα στοιχεία τους μπορούν να είναι διαφορετικού τύπου. Για παράδειγμα η πλειάδα `(true, 3u8)` αποτελεί πλειάδα δύο στοιχείων όπου το πρώτο στοιχείο είναι μία δυαδική τιμή και το δεύτερο ένας ακέραιος χωρίς πρόσημο.

Η πρόσβαση στις τιμές της πλειάδας γίνεται είτε αποδομώντας την πλειάδα με την έκφραση `let` είτε με την χρήση δεικτών εφόσον έχει δεθεί με κάποια μεταβλητή:

```
{
    let (boolean, num) = (true, 3u8);
    boolean; // true
    num // 3

    let tuple = (true, 3u8);
    tuple.0; // true
    tuple.1 // 3
}
```

2.2.2 Τελεστές

Οι τελεστές τις γλώσσας Rust για τις αριθμητικές, δυαδικές και λογικές εκφράσεις είναι παρόμοιοι με αυτούς της γλώσσας C και ακολουθούν την ίδια προτεραιότητα. Εξαίρεση αποτελούν οι τελεστές ++ και −− όπου δεν υπάρχουν στην γλώσσα Rust

2.2.3 Βασικές εκφράσεις και εντολές (expressions and statements)

Η γλώσσα Rust είναι κυρίως γλώσσα εκφράσεων, ωστόσο υπάρχουν δύο μορφές εντολών, οι εντολές δήλωσης και οι εντολές έκφρασης. Οι πρώτες χρησιμεύουν στην δήλωση αντικειμένων της γλώσσας όπως συναρτήσεις, τύπους κ.ά. και δεν επιστρέφουν κάποιο τύπο, ενώ οι δεύτερες χρησιμεύουν στην εκτέλεση εκφράσεων όπου χρειάζονται οι παρενέργειές τους αλλά όχι τα αποτελέσματά τους και επιστρέφουν τον μοναδιαίο τύπο. Οι εντολές δήλωσης εισάγονται με κάποια λέξη κλειδί ενώ οι εντολές έκφρασης αποτελούνται από την έκφραση ακολουθούμενη από ένα ερωτηματικό.

Δέσιμο μεταβλητών (Variables binding):

Το δέσιμο μεταβλητών είναι μια εντολή δήλωσης που μας επιτρέπει να αντιστοιχίσουμε μεταβλητές με τιμές και εισάγεται με την λέξη κλειδί `let`:

```
let pattern[: type] = bound_value;
```

ο τύπος είναι προαιρετικός εκτός από μερικές περιπτώσεις όπου δεν μπορεί να γίνει συμπερασμός τύπων και χρειάζεται η δήλωσή του. Όσον αφορά το μοτίβο (pattern) στην απλή μορφή του είναι ένα όνομα μεταβλητής:

```
let x = 5u32;
```

ωστόσο μπορεί να γίνει πιο σύνθετο και να πραγματοποιήσει αποδόμηση της τιμής στα μέρη αυτής δένοντας κάθε μέρος της σε μία μεταβλητή:

```
let (x, y) = (5u32, false);
```

Το δέσιμο μιάς μεταβλητής με την αντίστοιχη τιμή είναι αμετάβλητο ως προεπιλογή:

```
let x = true;
x = false; //Compile Error
```

ωστόσο αν χρειαζόμαστε ένα μεταβλητό δέσιμο μπορούμε να χρησιμοποιήσουμε την λέξη κλειδί `mut` που θα καταστήσει το δέσιμο μεταβλητό:

```
let mut x = true;
x = false;
```

Πριν την χρήση των μεταβλητών και συνήθως κατά την δήλωσή τους πρέπει να δένονται με μία τιμή (αρχικοποίηση) διαφορετικά θα προκληθεί σφάλμα κατά την μεταγλώττιση. Με αυτόν τον τρόπο αποφεύγεται η ανεπιθύμητη συμπεριφορά χρήσης μίας μεταβλητής η οποία δεν έχει αρχικοποιηθεί με κάποια τιμή.

Συναρτήσεις (Functions):

Κάθε πρόγραμμα γραμμένο στην γλώσσα Rust αποτελείται από τουλάχιστον μία συνάρτηση, την συνάρτηση `main()`:

```
fn main() { }
```

Η εντολή δήλωσης της συνάρτησης εισάγεται από την λέξη `fn` ακολουθούμενη από το όνομα της συνάρτησης, τις παραμέτρους της συνάρτησης χωριζόμενες με κόμμα εντός παρενθέσεων, τον τελεστή `->`, τον τύπο τον οποίο επιστρέφει η συνάρτηση και το σώμα της συνάρτησης εντός αγκύλων:

```
fn double(x: i32) -> i32 {
    x * 2
}
```

Σε περίπτωση που η συνάρτηση επιστρέφει τον μοανδιαίο τύπο μπορούμε να παραλείψουμε την δήλωσή του, δηλαδή το κομμάτι `-> ()`.

```
fn print_double(x: i32) {
    println!("{}", x * 2);
}
```

Το σώμα της συνάρτησης μπορεί να αποτελείται από εντολές ή εκφράσεις. Το αποτέλεσμα της συνάρτησης επιστρέφεται είτε αποτιμώντας την τελευταία έκφραση που θα εκτελεσθεί στο σώμα της συνάρτησης, είτε με την πρόωρη αποτίμηση και επιστροφή με την λέξη κλειδί `return` και την έκφραση:

```
fn double_if(x: i32) -> i32 {
    return x * 2;
}
```

Έλεγχος ροής (Flow control):

Για τον έλεγχο ροής η γλώσσα Rust χρησιμοποιεί παρόμοιες εκφράσεις με άλλες γλώσσες. Για τον έλεγχο συνθήκης έχουμε την έκφραση `if ... else if ... else`:

```
if boolean_expression {
    expression
} else if boolean_expression {
    expression
} else {
    expression
}
```

Τα `else if` και `else` είναι προαιρετικά, επίσης δεν χρειάζεται οι λογικές εκφράσεις να βρίσκονται εντός παρενθέσεων. Ως έκφραση το αποτέλεσμα της είναι κάποια τιμή η οποία πρέπει έχει τον ίδιο τύπο για κάθε διαφορετική ροή που μπορεί να ακολουθηθεί, διαφορετικά προκύπτει σφάλμα στην μεταγλώττιση.

Για τους επαναληπτικούς βρόχους υπάρχουν τρεις διαφορετικές εκφράσεις, `loop`, `while` και `for`. Η έκφραση `loop` υλοποιεί έναν ατέρμονα βρόχο:

```
loop {
    expression
}
```

Η έκφραση `while` υλοποιεί βρόχο που επαναλαμβάνεται όσο μία λογική έκφραση αποτιμάται ως αληθινή:

```
while boolean_expression {
    expression
}
```

Τέλος η έκφραση `for` υλοποιεί βρόχο με συγκεκριμένο αριθμό επαναλήψεων:

```
for variable in iterator {
    expression
}
```

Σε κάθε επανάληψη η μεταβλητή δένεται με την τιμή που επιστρέφει ο επαναλήπτης (iterator) μέχρι να εξαντληθούν τα στοιχεία του επαναλήπτη. Ως εκ τούτου ο αριθμός των επαναλήψεων είναι ίσος με το μέγεθος του επαναλήπτη.

Και στις τρεις περιπτώσεις υπάρχει η δυνατότητα διακοπής ή παράλειψης της επανάληψης χρησιμοποιώντας τις λέξεις `break` και `continue` αντίστοιχα:

```
loop {
    run_once();
    break;
}
```

Σε περίπτωση εμφωλιασμένων βρόχων επηρεάζεται ο πιο εσωτερικό βρόχος, εκτός αν χρησιμοποιηθούν ετικέτες βρόχων:

```
'outer for x in 0..10 {
    'inner for y in 0..10 {
        if y == 5 {
            continue 'outer
        }
    }
}
```

Ταίριασμα μοτίβων (Pattern Matching):

Η γλώσσα Rust υποστηρίζει το ταίριασμα μοτίβων:

```
match value {
    pattern => expression,
    pattern => expression,
    ...
    pattern => expression
}
```

Εφόσον η τιμή ταιριάξει με ένα από τα μοτίβα τότε εκτελείται η έκφραση που βρίσκεται στο δεξιό μέρος του τελεστή `=>`. Κατά την διάρκεια του ταιριάσματος η τιμή δύναται να αποδομηθεί και να δεθεί με μεταβλητές όπως γίνεται και με την έκφραση `let`.

Το ταίριασμα της τιμής πρέπει να γίνει αναγκαστικά με ένα από τα μοτίβα διαφορετικά θα προκληθεί σφάλμα στον μεταγλωττιστή και επειδή δεν είναι πάντα δυνατό να καταγράψουμε όλα τα πιθανά μοτίβα υπάρχει το μοτίβο `_` το οποίο ταιριάζει με οποιαδήποτε τιμή. Αυτό το μοτίβο μπορεί να χρησιμοποιηθεί και ως μέρος ενός μοτίβου έτσι ώστε να ταιριάξει με μία από τις τιμές:

```
match (3, true, 4) {
    (x, _, 4) => 4 + x,
    _ => 5
}
```

Ως μοτίβα μπορούμε να έχουμε κάποια εύρη τιμών όπως για παράδειγμα `1 ... 5` ή `'a' ... 'z'`. Στην περίπτωση αυτή μπορούμε να δέσουμε την τιμή με μία μεταβλητή:

```
match x {
    e @ 1 ... 5 => e,
    _ => 6
}
```

Πολλά μοτίβα μπορούν να αντιστοιχούν σε μία έκφραση με την βοήθεια του τελεστή `|` ή μπορεί να περιορίζονται από κάποια συνθήκη που εισάγεται με την λέξη κλειδί `if`:

```
match x {
    1 | 3 | 5 => 1,
    4 if y > 2 => 2,
    _ => 0
}
```

Τέλος υπάρχει η λέξη κλειδί `ref` την οποία την χρησιμοποιούμε μπροστά από την μεταβλητή του μοτίβου και δημιουργεί αναφορά σε αυτή. Η αναφορά στην μεταβλητή δεν μας επιτρέπει να αλλάξουμε την τιμή της εκτός και εάν το ορίσουμε ρητά χρησιμοποιώντας τις λέξεις κλειδιά `ref mut` μπροστά από την μεταβλητή.

2.2.4 Βασικοί Κατασκευαστές Τύπων

Απλή δομή (Struct):

Η απλή δομή επιτρέπει να προσθέσουμε διάφορους τύπους δεδομένων σε έναν ενιαίο τύπο. Η δήλωση της απλής δομής γίνεται με τον εξής τρόπο:

```
struct struct_name {
    field_name_1: field_type_1,
    field_name_2: field_type_2,
    ...
    field_name_n: field_type_n
}
```

Η πρόσβαση στα πεδία της απλής δομής γίνεται χρησιμοποιώντας το όνομα της μεταβλητής με την οποία έχει δεθεί η δομή ακολουθούμενο από μία τελεία και το όνομα του πεδίου:

```
{
    struct Point {
        x: f32,
        y: f32
    }

    let my_point = Point {x: 1.7, y: 2.5};
    my_point.x; // 1.7
}
```

Δομή πλειάδας (Tupple struct):

Πρόκειται για έναν συνδυασμό πλειάδας και απλής δομής:

```
struct struct_name(type_1, type_2, ... type_n)
```

Δομές πλειάδων με τον ίδιο αριθμό και τύπο στοιχείων αλλά διαφορετικό όνομα δεν θεωρούνται ίσες. Η πρόσβαση στα στοιχεία της γίνεται με τον ίδιο τρόπο που γίνεται στις πλειάδες:

```
{
    struct Point(f32, f32)

    let my_point = Point(1.7, 2.5);
    my_point.0; // 1.7
}
```

Μοναδιαία δομή (Unit-like struct):

Η μοναδιαία δομή είναι μία δομή χωρίς πεδία και χρησιμοποιείται σπάνια και κυρίως για να δηλώσει έναν νέο τύπο:

```
struct struct_name;
```

Δομή απαρίθμησης (Enum):

Η δομή αυτή επιτρέπει διαφορετικές παραλλαγές δομών να αντιπροσωπεύονται από έναν τύπο:

```
enum enum_name {
    struct_1,
    struct_2,
    ...
    struct_n
}
```

Οι παραλλαγές δομών μπορούν να είναι απλές δομές, δομές πλειάδας ή μοναδιαίες δομές και για να αναφερθούμε σε αυτές χρησιμοποιούμε την εξής σύνταξη:

```
enum_name::struct_n
```

2.2.5 Το Σύστημα ιδιοκτησίας/δανεισμού (Ownership/borrowing system)

Ένα από τα πιο σημαντικά χαρακτηριστικά της γλώσσας είναι το σύστημα ιδιοκτησίας/δανεισμού. Το σύστημα αυτό υλοποιεί την ασφάλεια μνήμης που είναι ένας από τους κύριους στόχους της γλώσσας. Επίσης αποτελεί παράδειγμα της μηδενικού κόστους αφαίρεσης που βοηθά στην αύξηση της ταχύτητας εκτέλεσης ενός προγράμματος γραμμένου σε Rust, καθώς η ανάλυση γίνεται κατά την διάρκεια της μεταγλώττισης και έτσι αποφεύγεται η χρήση συλλέκτη απορριφθέντων στοιχείων (garbage collector).

Ωστόσο το σύστημα δεν έρχεται χωρίς κόστος και αυτό είναι η αύξηση του χρόνου μεταγλώττισης του προγράμματος και η δυσκολία μάθησης του συστήματος. Πολλοί νέοι χρήστες μάχονται με τον ελεγκτή δανεισμού (borrow checker) όπου αποτελεί μέρος του μεταγλωττιστή καθώς καλούνται να λύσουν σφάλματα εκεί που θεωρούν ότι έχουν έγκυρο κώδικα. Το θετικό είναι ότι μετά από κάποιο διάστημα δουλεύοντας με τους κανόνες του συστήματος παρουσιάζονται όλο και λιγότερα σφάλματα από τον ελεγκτή δανεισμού.

Το σύστημα ιδιοκτησίας αποτελείται από τρεις βασικές ιδέες της οποίες και θα δούμε αναλυτικά:

- Ιδιοκτησία (Ownership)
- Δανεισμός (Borrowing)
- Διάρκεια ζωής (Lifetime)

Ιδιοκτησία (Ownership):

Η ιδέα της ιδιοκτησίας είναι ότι κάθε πόρος του προγράμματος έχει έναν μοναδικό ιδιοκτήτη. Η ιδιοκτησία ενός πόρου επιτρέπει στον ιδιοκτήτη της να έχει πρόσβαση σε αυτόν, να τον μεταβιβάσει, να τον δανείσει ή να τον καταστρέψει όσο βρίσκεται στην κατοχή του.

Ένας τρόπος να δοθεί η ιδιοκτησία ενός πόρου σε ένα ιδιοκτήτη είναι κατά το δέσιμο μεταβλητής με το `let`:

```
{
    let x = HashMap::new(); //x now owns the new Hashmap
}
```

Με το παραπάνω δέσιμο πλέον η μεταβλητή `x` είναι ο ιδιοκτήτης του χάρτη κατακερματισμού (Hashmap). Ως ιδιοκτήτης έχει πρόσβαση στα δεδομένα του χάρτη, που αναλόγως αν τα δεδομένα αυτά είναι μεταβλητά ή αμετάβλητα η πρόσβαση μπορεί να μεταφραστεί ως ανάγνωση ή ανάγνωση και εγγραφή αντίστοιχα.

Η μεταφορά της ιδιοκτησίας μπορεί να γίνει με διάφορους τρόπους, για παράδειγμα κατά το δέσιμο μεταβλητής ή κατά την κλήση μίας συνάρτησης:

```
{
    let x = HashMap::new();
    let y = x //transfering ownership from x to y
    give(y) //transfering ownership from y to function give
}
```

Μετά την μεταφορά της ιδιοκτησίας ο (πρώην) ιδιοκτήτης παύει να έχει δικαιώματα πάνω στον πόρο που μεταβίβασε. Έτσι στο παραπάνω παράδειγμα μετά την μεταφορά της ιδιοκτησίας από τη μεταβλητή `x` στην `y`, η `x` δεν μπορεί να χρησιμοποιηθεί ώστε να εισάγουμε, για παράδειγμα, ένα καινούριο στοιχείο στον χάρτη. Αντίστοιχα μετά τη μεταβίβαση του χάρτη από την `y` στη συνάρτηση `give`, δεν μπορεί να χρησιμοποιηθεί και η `y`.

Σε περίπτωση που επιχειρήσουμε να χρησιμοποιήσουμε τις μεταβλητές θα υπάρξει σφάλμα στην μεταγλώττιση. Η μόνη περίπτωση που δεν θα προκληθεί σφάλμα είναι αν ο πόρος υλοποιεί το γνώρισμα της αντιγραφής (Copy trait). Σε αυτή την περίπτωση γίνεται αντιγραφή του πόρου και έτσι κάθε μεταβλητή είναι πλέον ιδιοκτήτης διαφορετικού αντιγράφου. Παράδειγμα τύπων που υλοποιούν το

γνώρισμα της αντιγραφής είναι όλοι οι βασικοί τύποι. Έτσι μεταγλωττίζοντας το παρακάτω κώδικα δεν θα υπάρξει κάποιο σφάλμα.

```
{
    let x = true;
    let y = x
    println!("{} — {}", x, y) //true — true
}
```

Σε πολλές περιπτώσεις η ιδέα της ιδιοκτησίας μας περιορίζει αρκετά και δεν είναι πάντα δυνατό από πλευράς χωρικού ή χρονικού κόστους να αντιγράφουμε τους πόρους του προγράμματος. Για αυτό υπάρχει η ιδέα του δανεισμού.

Δανεισμός (Borrowing):

Ο ιδιοκτήτης μπορεί να δανείσει το πόρο που έχει στην διάθεσή του σε έναν άλλο προσωρινό ιδιοκτήτη. Όσο διαρκεί ο δανεισμός ο προσωρινός ιδιοκτήτης αποκτά δικαιώματα πρόσβασης και επιπλέον δανεισμού πάνω στον πόρο. Όταν ο προσωρινός ιδιοκτήτης παύει να υπάρχει ο πόρος επιστρέφεται στον αρχικό ιδιοκτήτη.

Ο δανεισμός γίνεται με την προϋπόθεση ότι ο πόρος θα επιστραφεί στον αρχικό ιδιοκτήτη πριν αυτός πάψει να υπάρχει και καταστραφεί ο πόρος. Σε διαφορετική περίπτωση ο ελεγκτής δανεισμού θα προκαλέσει σφάλμα κατά την μεταγλώττιση.

Υπάρχουν δύο είδη δανεισμού ο ένας είναι ο δανεισμός διαμοιρασμού (shared borrow) όπου η πρόσβαση του προσωρινού ιδιοκτήτη περιορίζεται στην ανάγνωση και ο δεύτερος είναι ο δανεισμός μεταβολής (mutable borrow) όπου η πρόσβαση μπορεί να είναι και ανάγνωση και εγγραφή.

Κατά την διάρκεια του δανεισμού διαμοιρασμού ο ιδιοκτήτης μπορεί να δανείσει με τον ίδιο τρόπο σε περισσότερους του ενός ιδιοκτήτες τον πόρο ταυτόχρονα. Επίσης, η πρόσβαση του ιδιοκτήτη όσο διαρκεί ο δανεισμός περιορίζεται μόνο στην ανάγνωση και μην έχοντας στην κατοχή του τον πόρο δεν μπορεί να τον καταστρέψει ή να τον μεταβιβάσει. Ο δανεισμός διαμοιρασμού γίνεται με τον τελεστή & και ονομάζεται αναφορά (reference):

```
{
    let x = true;
    let y = &x;
    println!("y is shared borrow of x with value {}", y)
}
```

Στην περίπτωση του δανεισμού μεταβολής ο αρχικός ιδιοκτήτης δεν μπορεί να δανείσει ή να έχει οποιαδήποτε πρόσβαση στον πόρο μέχρι αυτός να επιστραφεί από τον προσωρινό ιδιοκτήτη. Για τον δανεισμό μεταβολής χρησιμοποιείται ο τελεστής & όπου ακολουθείται από την λέξη κλειδί mut και ονομάζεται αναφορά μεταβολής (mutable reference). Για να γίνει μεταβολή στον πόρο χρησιμοποιείται ο τελεστής * έτσι ώστε να έχουμε πρόσβαση στην τιμή της αναφοράς:

```
{
    let mut x = true;
    let y = &mut x;
    *y = false;
    println!("y is mutable borrow of x with value {}", y)
}
```

Η ιδέα του δανεισμού έτσι όπως υλοποιείται εξασφαλίζει σε χρόνο μεταγλώττισης ότι δεν θα υπάρχει χρήση ενός πόρου μετά την καταστροφή του, ούτε ταυτόχρονη πρόσβαση από παραπάνω του ενός ιδιοκτήτες σε περίπτωση που ο πόρος δύναται να μεταβληθεί.

Διάρκεια ζωής (Lifetime):

Η ιδέα της διάρκειας ζωής έρχεται να υλοποιήσει την προϋπόθεση όπου ο προσωρινός ιδιοκτήτης πρέπει να επιστρέψει τον πόρο που έχει δανειστεί πριν ο αρχικός ιδιοκτήτης του τον καταστρέψει. Αυτό γίνεται συγκρίνοντας την διάρκεια ζωής του ιδιοκτήτη με την διάρκεια ζωής του προσωρινού ιδιοκτήτη που ταυτίζεται με την διάρκεια ζωής του δανεισμού.

Η διάρκεια ζωής μιας μεταβλητής ξεκινάει την στιγμή που αυτή δένεται με κάποια τιμή και σταματά όταν η μεταβλητή βγει εκτός εμβέλειας και καταστρέφεται. Παρόμοια γίνεται και με τις αναφορές, ωστόσο σε κάποιες περιπτώσεις ο μεταγλωττιστής δεν μπορεί να συμπεράνει την διάρκεια ζωής. Σε αυτές τις περιπτώσεις ο προγραμματιστής πρέπει να δηλώσει ρητά την διάρκεια ζωής με μία παράμετρο. Χαρακτηριστικά παραδείγματα είναι δομές τύπων που φέρουν ως πεδία αναφορές και ορισμοί συναρτήσεων που στις υπογραφές τους εμπλέκονται αναφορές.

Η δήλωση της διάρκειας ζωής γίνεται με τον χαρακτήρα ’ και ακολουθεί το όνομα της παραμέτρου, που συνηθίζεται να είναι ένα πεζό γράμμα, το οποίο δένεται με την διάρκεια. Παρακάτω φαίνονται δύο παραδείγματα δήλωσης διάρκειας ζωής για μία απλή δομή και για μία συνάρτηση:

```
struct BorrowedContent<’a> {
    x: &’a i32
}

fn function<’a, ’b>( int1: &’a i32,
    int2: &’a i32, bool: &’b bool ) —> &’a i32
{
    if *bool { int1 } else { int2 }
}
```

Ειδικά για τις υπογραφές των συναρτήσεων υπάρχει η διαδικασία συμπερασμού διάρκειας ζωής (lifetime elision) όπου μας βοηθά να εξαλειφθούν οι δηλώσεις σε αρκετές περιπτώσεις και διευκολύνει την αναγνωσιμότητα του κώδικα. Για να γίνει ο συμπερασμός και η εξάλειψη ακολουθούνται οι παρακάτω τρεις κανόνες:

- Κάθε διάρκεια ζωής που εξαλείφεται από τις παραμέτρους της συνάρτησης γίνεται παράμετρος διάρκειας ζωής της συνάρτησης.
- Αν υπάρχει μόνο μία παράμετρος διάρκειας ζωής στην συνάρτηση αυτή δένεται με όλες τις τιμές που επιστρέφει η συνάρτηση.
- Αν υπάρχουν πολλές παράμετροι διάρκειας ζωής στην συνάρτηση και μία από αυτές προέρχεται από τις αναφορές `&self` ή `&mut self` τότε αυτή η παράμετρος δένεται με τις τιμές που επιστρέφει η συνάρτηση.

Αν κανένας από τους παραπάνω κανόνες δεν ικανοποιείται τότε προκύπτει σφάλμα και ο προγραμματιστής θα πρέπει να δηλώσει ρητά τις διάρκειες ζωής, τέτοιο παράδειγμα αποτελεί το παράδειγμα παραπάνω.

Τέλος υπάρχει η στατική διάρκεια ζωής (`’static`) η οποία ταυτίζεται με την διάρκεια ζωής του προγράμματος. Υπάρχουν μόνο δύο τρόποι μία μεταβλητή να έχει στατική διάρκεια ζωής, ο πρώτος είναι όταν δένεται με μια αλφαριθμητική παράσταση και η δεύτερη όταν δένεται με μία στατική σταθερά:

```
{
    let x : &’static str = ”variable with static lifetime”;

    static NUMBER: i32 = 42;
    let y: &’static i32 = &NUMBER;
}
```

2.2.6 Συντακτικό μεθόδου (Method Syntax)

Η Rust υποστηρίζει το συντακτικό κλήσης μεθόδων (method syntax call):

```
item.function_1().function2() // function_2(function_1(item))
```

Για την δήλωση των μεθόδων που αντιστοιχούν σε έναν τύπο χρησιμοποιείται η λέξη κλειδί `impl` και ακολουθείται από τις δηλώσεις των μεθόδων εντός αγκύλων:

```
struct Measurement {
    length: f64,
    width: f64,
    height: f64,
}
impl Measurement {
    fn volume(&self) -> f64 {
        (self.length * self.width * self.height)
    }
}
fn main() {
    let m = Measurement { length: 2.0, width: 3.0, height: 7.0 };
    println!("{}", m.volume());
}
```

Η πρώτη παράμετρος μίας μεθόδου είναι το αντικείμενο για το οποίο υλοποιείται η μέθοδος και μπορεί να έχει τρεις μορφές δήλωσης:

- `self`: όπου το αντικείμενο μεταβιβάζεται στην μέθοδο.
- `&self`: όπου η μέθοδος δανείζεται το αντικείμενο.
- `&mut self`: όπου η μέθοδος δανείζεται το αντικείμενο με δικαίωμα μεταβολής του.

Εκτός από μεθόδους μπορούν να δηλωθούν συναρτήσεις που σχετίζονται με το αντικείμενο που υλοποιείται (associated functions). Οι συναρτήσεις αυτές δεν καλούνται με τον τρόπο των μεθόδων αλλά με το όνομα του αντικειμένου, τον τελεστή `::` και την κλήση της συνάρτησης. Σε άλλες γλώσσες οι συναρτήσεις αυτές είναι γνωστές και ως στατικές μέθοδοι:

```
struct Measurement {
    length: f64,
    width: f64,
    height: f64,
}
impl Measurement {
    fn measure(length: f64, width: f64, height: f64) -> Measurement
    {
        Measurement {
            length: length,
            width: width,
            height: height
        }
    }
}
fn main() {
    let m = Measurement::measure(2.0, 3.0, 7.0 };
    println!("{}", m.length * m.width * m.height);
}
```

2.2.7 Γενικοί Τύποι (Generics)

Στα πλαίσια της μηδενικού κόστους αφαίρεσης η Rust υλοποιεί τον παραμετρικό πολυμορφισμό με τους γενικούς τύπους. Οι γενικοί τύποι συναντώνται στις δηλώσεις κατασκευαστών τύπων και συναρτήσεων και δηλώνονται μετά το όνομα του κατασκευασμένου τύπου ή της συνάρτησης ανάμεσα στους χαρακτήρες < και > συνήθως με ένα κεφαλαίο γράμμα. Έπειτα χρησιμοποιούνται στο σώμα του κατασκευασμένου τύπου ή της συνάρτησης στην θέση των τύπων:

```
struct Measurement <T>{
    length: T,
    width: T,
    height: T,
}
fn measure<T>(length: T, width: T, height: T) -> Measurement<T> {
    Measurement {
        length: length,
        width: width,
        height: height
    }
}
fn main() {
    let simple_measurement = measure(2_i32, 3_i32, 7_i32);
    let precise_measurement = measure(2.0_f32, 3.0_f32, 7.0_f32);
}
```

Ένας από τους πιο γνωστούς τύπους της τυπικής βιβλιοθήκης (standart library) της Rust είναι ο τύπος `Option<T>` ο οποίος βασίζεται στους γενικούς τύπους. Πρόκειται για την απαρίθμηση (enumeration) που είτε έχει κάποια τιμή τύπου `T` είτε δεν έχει καμία:

```
enum Option<T> {
    Some(T),
    None,
}
```

Υπάρχουν περιπτώσεις όπου χρειάζεται να ορίσουμε ρητά τον τύπο με τον οποίο ταυτίζεται ο γενικός τύπος. Αν η περίπτωση αυτή αφορά δήλωση τύπου, αντικαθίσταται το όνομα του γενικού τύπου με τον τύπο που θέλουμε. Αν πρόκειται για κλήση συνάρτησης τότε μετά το όνομα της συνάρτησης ακολουθεί ο τελεστής `::` και έπειτα ο τύπος που θέλουμε εντός των χαρακτήρων < και >:

```
fn main() {
        let x: Option<i32> = None;

        fn is_some<T>(x: Option<T>) -> bool {
            match x {
                Some(_) => true,
                None => false
            }
        }

        println!("{}", is_some::<i32>(x))
}
```

Για την επιλογή της κατάλληλης συνάρτησης για τον κάθε τύπο κατά την διάρκεια εκτέλεσης ενός προγράμματος, η Rust χρησιμοποιεί στατική επιλογή (static dispatch) έτσι ώστε να μην επιβαρύνει χρονικά την εκτέλεση.

2.2.8 Γνωρίσματα (Traits)

Τα γνωρίσματα χρησιμοποιούνται στην Rust ώστε να ορίσουμε την λειτουργικότητα που πρέπει να υλοποιεί ένας τύπος. Η δήλωση των γνωρισμάτων γίνεται με την λέξη κλειδί `trait`, το όνομα του γνωρίσματος και το σώμα με τις δηλώσεις των υπογραφών (signatures) των μεθόδων ή των συναρτήσεων που είναι προς υλοποίηση:

```
trait HasVolume {
    fn volume(&self) -> f64;
}
```

Η υλοποίηση ενός γνωρίσματος γίνεται με παρόμοιο τρόπο με την υλοποίηση ενός τύπου. Εισάγεται και αυτή με την λέξη κλειδί `impl` και ακολουθείται από το όνομα του γνωρίσματος, την λέξη κλειδί `for` και τον τύπο που υλοποιεί το γνώρισμα:

```
struct Measurement{
    length: f64,
    width: f64,
    height: f64,
}

trait HasVolume {
    fn volume(&self) -> f64;
}

impl HasVolume for Measurement{
    fn volume(&self) -> f64 {
        (self.length * self.width * self.height)
    }
}
```

Στο σώμα της δήλωσης του γνωρίσματος μπορούμε να δηλώσουμε και σώμα στις συναρτήσεις και τις μεθόδους. Αυτό θα χρησιμοποιηθεί ως προεπιλεγμένη υλοποίηση για όσους τύπους υλοποιούν το γνώρισμα. Ωστόσο αν για κάποιο τύπο χρειάζεται άλλη συμπεριφορά από την προεπιλεγμένη τότε μπορεί να δηλώσει την μέθοδο ή την συνάρτηση με την επιθυμητή συμπεριφορά και αυτή θα αντικαταστήσει την προεπιλεγμένη.

Μία άλλη χρήση των γνωρισμάτων είναι ο περιορισμός των γενικών τύπων. Για παράδειγμα σε μία συνάρτηση γενικού τύπου όπου υπάρχει η πράξη της σύγκρισης υπάρχουν τύποι που δεν μπορούν να την εκτελέσουν έτσι ο μεταγλωττιστής θα προκαλέσει σφάλμα. Περιορίζοντας όμως τον γενικό τύπο μόνο σε αυτούς τους τύπους που υλοποιούν την σύγκριση με το γνώρισμα `Ord` τότε το πρόγραμμα μεταγλωττίζεται κανονικά:

```
fn is_greater<T: Ord>(x: T, y: T) -> bool {
    x > y
}
```

Αν θέλουμε να περιορίσουμε με παραπάνω από ένα γνώρισμα έναν γενικό τύπο τότε στην δήλωση χρησιμοποιούμε τον τελεστή `+` και προσθέτουμε τα γνωρίσματα.

```
fn is_greater<T: Ord + Eq>(x: T, y: T) -> bool {
    x > y
}
```

Ένα γνώρισμα μπορεί να εξαρτάται από ένα άλλο γνώρισμα, αυτό σημαίνει ότι η υλοποίηση του εξαρτόμενου γνωρίσματος προϋποθέτει την υλοποίηση του γνωρίσματος από το οποίο εξαρτάται. Με αυτόν τον τρόπο υλοποιείται ένα είδος κληρονομικότητας. Για να δηλώσουμε την εξάρτηση ενός

γνωρίσματος από ένα άλλο κατά την δήλωσή του μετά από το όνομα ακολουθεί άνω κάτω τελεία και έπειτα το γνώρισμα από το οποίο εξαρτάται:

```
trait Vehicle {
    fn speed(&self) -> f64;
}

trait Car: Vehicle {
    fn doors(&self) -> i32;
}
```

2.2.9 Ανώνυμες συναρτήσεις (Closures)

Η Rust υποστηρίζει ανώνυμες συναρτήσεις που έχουν την παρακάτω γενική μορφή:

```
|parameters| expression
```

Οι παράμετροι της ανώνυμης συνάρτησης μπορεί να είναι καμία, μία ή περισσότερες χωριζόμενες με κόμμα. Σε περιπτώσεις όπου ο συμπερασμός τύπων δεν μπορεί να συμπεράνει τους τύπους των παραμέτρων ή του απότελέσματος τότε πρέπει να δηλωθούν ρητά:

```
|x: i32, y: i32| -> Option<i32> { None };
```

Οι ανώνυμες συναρτήσεις ως προεπιλογή μπορούν να δανειστούν μεταβλητές από το περιβάλλον τους είτε με αναφορά είτε με αναφορά μεταβολής:

```
{
    let num = 17;
    let plus_num = |x: i32| x + num;

    let mut mutable_num = 35;
    let mut add_to_num = |x: i32| mutable_num += x;
}
```

Στην περίπτωση όπου η ανώνυμη συνάρτηση χρειάζεται να έχει την ιδιοκτησία των μεταβλητών του περιβάλοντός της τότε το δηλώνουμε ρητά χρησιμοποιώντας την λέξη κλειδί `move` και ισχύουν οι κανόνες μεταβίβασης ιδιοκτησίας:

```
{
    let num = 17;
    let plus_num = move |x: i32| x + num;

    let mut mutable_num = 35;
    let mut add_to_num = move |x: i32| mutable_num += x;
}
```

Η κλήση μίας ανώνυμης συνάρτησης γίνεται με το όνομα της μεταβλητής που έχει δεθεί μαζί της ακολουθούμενη από τις παραμέτρους εντός παρενθέσεων:

```
{
    let num = 17;
    let plus_num = |x: i32| x + num;
    println!("{}" plus_num(25));
}
```

2.2.10 Διάνυσμα (Vector)

Ένας από τους συχνά χρησιμοποιούμενους τύπους της τυπικής βιβλιοθήκης είναι το διάνυσμα. Το δυνανυσμα είναι η υλοποίηση ενός δυναμικού πίνακα και έχει τύπο `Vec<T>`, όπου `T` είναι γενικός τύπος. Ένα διάνυσμα μπορεί να οριστεί με την μακροεντολή vec! ή με την συνάρτηση `new()`:

```
{
    let mut v1 = vec![1, 2, 3, 4];
    let mut v2 = vec![0; 4];
    let mut v3: Vec<i32> = Vec::new();
}
```

Καθώς το διάνυσμα αποτελεί δυναμική δομή δεδομένων, αυτό σημαίνει ότι δεσμέυει δυναμικά νέα μνήμη κατά τον χρόνο εκτέλεσης κάθε φορά που έχει γεμίσει τον δεσμευμένο του χώρο. Για αυτόν τον λόγο δίνεται η δυνατότητα αν είναι γνωστό το μέγεθος της μνήμης που θα χρειαστεί να το δηλώσουμε κατά την δημιουργία του με την συνάρτηση `with_capacity`:

```
{
    let v3: Vec<i32> = Vec::with_capacity(100);
}
```

Η προσθήκη και η αφαίρεση στοιχείων γίνεται με τις μεθόδους push και pop οι οποίες προσθέτουν και αφαιρούν από το τέλος του δανύσματος μία τιμή, ενώ η πρόσβαση στις τιμές του διανύσματος γίνεται με τον ίδιο τρόπο που γίνεται στον πίνακα:

```
{
    let mut v = vec![1, 2];
    v.push(3);
    v.push(4);
    println!(”{}”, v[1]); // 2
    println!(”{}”, v.pop().unwrap()); // 4
}
```

Ένας άλλος τρόπος εισαγωγής στοιχείου σε συγκεκριμένο σημείο και αφαίρεσης στοιχείου από συγκεκριμένο σημείο γίνεται με τις μεθόδους `insert` και `remove` αντίστοιχα:

```
{
    let mut v = vec![1, 2];
        v.insert(0,3);
        println!(”{}”, v[1]); // 1
        println!(”{}”, v.remove(2)); // 2
}
```

Τέλος μία βασική λειτουργία του διανύσματος είναι ότι μπορεί να χρησιμοποιηθεί ως επαναλήπτης πανω στις τιμές του:

```
{
    let v = vec![1, 2, 3, 4];
    for i in v {
       println!(”{}”, i)
    }
}
```

2.2.11 Συμβολοσειρά (String)

Η δομή της συμβολοσειράς ανήκει στην τυπική βιβλιοθήκη και διαφέρει από τον βασικό τύπο `str` καθώς πρόκειται για μία δυναμική δομή UTF-8 χαρακτήρων:

```
{
    let s1 = ”This is a String”.to_string();
    let static_str: &’static str = ”str”;
    let s2 = static_str.to_string();
}
```

Ο δυναμικός χαρακτήρας της δομής μας επιτρέπει να συνδέσουμε στο τέλος μίας συμβολοσειράς μία βασικού τύπου συμβολοσειρά:

```
{
    let s1 = ”Hello, ”.to_string();
    let static_str: &’static str = ”world!”;
    let s2 = s1 + static_str;
}
```

Καθώς η συμβολοσειρά είναι μία σειρά χαρακτηρων UTF-8 αυτό σημαίνει ότι ένας χαρακτήρας μπορεί να αντιστοιχεί σε παραπάνω από ένα bytes και ως εκ τούτου η πρόσβαση σε έναν χαρακτήρα με τον τρόπο πρόσβασης που ισχύει στους πίνακες δεν έχει νόημα και προκαλεί σφάλμα.

Ωστόσο μπορεί να χρησιμοποιηθεί η μέθοδος `chars` που επιστρέφει έναν επαναλήπτη πάνω στους χαρακτήρες της συμβολοσειράς:

```
{
    let s = ”ABC”.to_string();
    for i in s.chars() {
        print!(”{}” i)
    } // ABC
}
```

2.2.12 Βασικές Μακροεντολές

Η Rust διαθέτει μηχανισμό δημιουργίας μακροεντολών και η χρήση τους ξεχωρίζει από το θαυμαστικό που χρησιμοποιούν, μερικές από τις πιο κοινές μακροεντολές είναι οι εξής:

- **`panic!`:** δημιουργεί σφάλμα στο τρέχον νήμα. Μπορεί να συνοδευτεί και από κάποιο μήνυμα. Παράδειγμα: `panic!(”Panic!!!”)`
- **`vec!`:** δημιουργεί ένα διάνυσμα με τις τιμές που περικλείονται εντός τετράγωνων αγκύλων. Παράδειγμα: `vec![1, 2,3]`
- **`assert!`:** χρησιμοποιείται σε κατάσταση δοκιμών και αν η έκφραση που περικλείεται στις παρενθέσεις είναι `true` τότε περνάει επιτυχώς την δοκιμή αλλιώς προκαλείται `panic!`. Παράδειγμα: `assert!(1 == 1)`
- **`assert_eq!`:** ίδια με την `assert!` αλλά λαμβάνει δύο ορίσματα και ελέγχει την ισότητά τους. Παράδειγμα: `assert_eq!(3, 2+1)`
- **`unreachable!`:** χρησιμοποιείται σε σημεία που θεωρητικά η ροή του προγράμματος δεν θα φτάσει ποτέ. Παράδειγμα:

```
if false {
    unreachable!();
}
```

2.3 Διαχειριστής Cargo

Ο διαχειριστής Cargo αποτελεί ένα εργαλείο που βοηθά στην διαχείριση των πρότζεκτ της Rust και των εξαρτήσεών τους. Η βασική λειτουργία του είναι κατά το χτίσιμο ενός πρότζεκτ να ελέγχει αν έχουν δηλωθεί κάποιες εξαρτήσεις, να τις συγκεντρώνει και να τις χτίζει, και έπειτα να προχωρά στο χτίσιμο του πρότζεκτ.

Η παραμετροποίηση του εργαλείου γίνεται στο αρχείο Cargo.toml όπου εκεί δηλώνονται βασικά στοιχεία για το πρότζεκτ όπως το όνομα του, η έκδοσή του και οι δημιουργοί του, οι εξαρτήσεις του πρότζεκτ και οι λεπτομέρειες αυτών όπως από που και ποια έκδοσή του θα μεταφορτωθεί. Επίσης δηλώνονται διάφορες παράμετροι σχετικά με το χτίσιμο και το περιβάλλον χτισίματος του πρότζεκτ. Η διάταξη ενός πρότζεκτ που διαχειρίζεται ο διαχειριστής Cargo έχει την εξής μορφή:

- **Cargo.toml και Cargo.lock:** τοποθετούνται στην ρίζα του πρότζεκτ και αποθηκεύουν όλα τα μεταδεδομένα που σχετίζονται με το πρότζεκτ.
- **Κατάλογος src/:** περιέχει των πηγαίο κώδικα του πρότζεκτ.
- **Αρχείο src/lib.rs:** το προεπιλεγμένο αρχείο πρότζεκτ που παράγει βιβλιοθήκη.
- **Αρχείο src/lib.rs:** το προεπιλεγμένο αρχείο πρότζεκτ που παράγει εκτελέσιμο.
- **Κατάλογος src/bin/:** περιέχει πρόσθετα εκτελέσιμα αρχεία.
- **Κατάλογος tests/:** περιέχει τα αρχεία δοκιμών
- **Κατάλογος examples/:** περιέχει παραδείγματα χρήσης
- **Κατάλογος bench/:** περιέχει αρχεία δοκιμών επίδοσης (benchmarks)

Εκτός όμως από την διαχείριση του χτισίματος του πρότζεκτ, ο διαχειριστής Cargo μπορεί να αναλάβει και την διαχείριση των δοκιμών του πρότζεκτ, αναζητώντας δοκιμές (tests) στα αρχεία του και εκτελώντας τες επιστρέφοντας αναλυτική αναφορά με τα αποτελέσματα. Οι δοκιμές που εκτελούνται είναι τεσσάρων ειδών και βρίσκονται:

- Εντός των αρχείων του κώδικα του πρότζεκτ, όπου γράφονται οι δοκιμές μονάδων (unit tests)
- Σε ξεχωριστά αρχεία εντός του καταλόγου tests, όπου γράφονται οι δοκιμές ενοποίησης (integration tests)
- Εντός της τεκμηρίωσης ως παραδείγματα κώδικα.
- Σε ξεχωριστά αρχεία εντός του καταλόγου examlpes, όπου βρίσκονται τα παραδείγματα χρήσης

Κεφάλαιο 3

Χρόνος - Ημερομηνία και Ώρα

3.1 Βασικές Έννοιες

3.1.1 Ρολόι - Ώρα

Το ρολόι είναι το σύστημα μέτρησης και οργάνωσης του χρόνου από το επίπεδο της ημέρας προς μικρότερα επίπεδα, κοινώς γνωστού και ως ώρα. Η ιστορία του ρολογιού ξεκινά από την ανάγκη του ανθρώπου να υποδιαιρέσει την διάρκεια της ημέρας. Σαν συσκευή μέτρησης του χρόνου το ρολόι έχει περάσει από πολλά στάδια με διαφορετικές υποδιαιρέσεις και ακρίβεια καθώς και διάφορους τρόπους κατασκευής.

Σήμερα διεθνώς για την μέτρηση της ώρας χρησιμοποιείται το ατομικό ρολόι, ωστόσο επειδή αποτελεί συσκευή ακριβείας και έχει σημαντικό κόστος κατασκευής υπάρχουν λίγα παγκοσμίως. Ο περισσότερος κόσμος χρησιμοποιεί τα ηλεκτρονικά και τα μηχανικά ρολόγια ενώ είναι διαδεδομένα και οι ηλεκτρονικοί δέκτες που λαμβάνουν ραδιοσήματα με την ατομική ώρα και την προβάλουν.

Δευτερόλεπτο

Αν και παλιότερα υπήρχε η έννοια της υποδιαίρεσης της ημέρας, μόλις το 1000 μ.Χ. έγινε η χρήση του όρου δευτερόλεπτο ως υποδιαίρεση. Μέχρι τότε υπήρχαν διάφοροι πολιτισμοί που χρησιμοποιούσαν υποδιαιρέσεις βασισμένες στον αριθμό 60, ωστόσο δεν είχαν προσεγγίσει το σημερινό δευτερόλεπτο. Το 1267 συναντούμε ακόμα μία αναφορά στην υποδιαίρεση του δευτερολέπτου. Τα πρώτα ρολόγια που απεικόνιζαν το δευτερόλεπτο εμφανίστηκαν στο δεύτερο μισό του 16ου αιώνα κάνοντας χρήση ελασμάτων, ενώ μία πιο ακριβής μέτρηση του δευτερολέπτου έγινε περίπου έναν αιώνα μετά με την εμφάνιση των ρολογιών τύπου εκκρεμές.

Η πρώτος ορισμός του δευτερολέπτου με βάση το έτος έγινε το 1956 και βασίστηκε σε κλάσμα της διάρκειας του ηλιακού/ ή τροπικού έτους 1900 και επισημοποιήθηκε το 1960 από την Γενική Διάσκεψη Μέτρων και Σταθμών (CGPM) μαζί με την καθιέρωση του διεθνούς συστήματος μέτρησης. Ωστόσο καθώς είχε παρατηρηθεί ότι η κίνηση γης-ηλίου δεν ήταν σταθερή και καθώς είχαν κάνει την εμφάνισή τους τα ατομικά ρολόγια, 7 χρόνια μετά ο ορισμός του δευτερολέπτου άλλαξε στην διάρκεια 9.192.631.770 περιόδων ακτινοβολίας από μετάπτωση μεταξύ δύο επιπέδων ενέργειας του ατόμου του Καισίου-133.

Λεπτό

Το λεπτό είναι η πρώτη υποδιαίρεση της ώρας σε 60 μέρη και είναι ιστορικά συνδεμένη με το δευτερόλεπτο. Σήμερα το λεπτό αποτελείται από 60 δευτερόλεπτα και 60 λεπτά αποτελούν μία ώρα.

Ώρα

Η ώρα ως υποδιαίρεση της μέρας εμφανίζεται στους αρχαίους πολιτισμούς και συνδέεται με τα πρώτα ηλιακά ρολόγια. Η ώρα αποτελείται από 60 λεπτά ή 3600 δευτερόλεπτα και 24 ώρες αποτελούν μία ημέρα. Η σημερινή μέτρηση της ώρας αρχίζει τα μεσάνυκτα και αν το ρολόι είναι 12ωρο επανεκκινεί το μεσημέρι ενώ αν είναι 24ωρο επανεκκινεί ξανά τα μεσάνυχτα.

3.1.2 Ημερολόγιο - Ημερομηνία

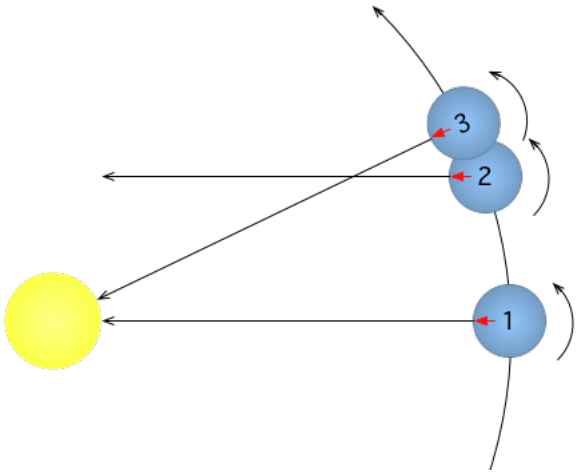
Το ημερολόγιο είναι το σύστημα μέτρησης και οργάνωσης του χρόνου από το επίπεδο της ημέρας προς μεγαλύτερα επίπεδα, κοινώς γνωστού και ως ημερομηνία. Η ιστορία των ημερολογίων συνδέεται από την εμφάνισή τους μέχρι και σήμερα με διάφορα αστρονομικά φαινόμενα. Τα κυριότερα από αυτά είναι η περιστροφή της γης γύρω από τον άξονά της, η κίνηση και οι φάσεις της σελήνης και η κίνηση της γης γύρω από τον ήλιο. Σήμερα διεθνώς χρησιμοποιείται το Γρηγοριανό ημερολόγιο, ωστόσο σε κάποια μέρη του κόσμου χρησιμοποιούν για κοινωνικούς, θρησκευτικούς ή/και πολιτικούς λόγους και άλλα ημερολόγια.

Ημέρα

Η ημέρα ορίζεται ως ο χρόνος μια πλήρους περιστροφής της γης γύρω από τον άξονά της, ιδανικά και όπως επισήμως μετριέται σήμερα από το διεθνές σύστημα αντιστοιχεί σε 24 ώρες ή 86400 δευτερόλεπτα. Ωστόσο στην πράξη λόγω φυσικών φαινομένων, όπως η παλίρροια, το χρονικό διάστημα δεν είναι σταθερό. Επίσης το χρονικό διάστημα είναι διαφορετικό ανάλογα με το πιο είναι το σημείο αναφοράς με το οποίο μετράμε την περιστροφή.

Στην ηλιακή μέρα, δηλαδή με σημείο αναφοράς τον ήλιο, έχουμε την μέρα να διαρκεί κατά μέσο όρο 86400,002 δευτερόλεπτα τις τελευταίες δεκαετίες, ενώ στην αστρική μέρα, με σημείο αναφοράς κάποιο μακρινό άστρο, θεωρώντας το ως σταθερό σημείο, η μέρα διαρκεί 86154,1 δευτερόλεπτα, περίπου 4 λεπτά λιγότερα από την ηλιακή.

Σχήμα 3.1: Αστρική Ημέρα



Source: ”Sidereal day (prograde)”. Licensed under CC BY-SA 3.0 via Commons - https://commons.wikimedia.org/wiki/File:Sidereal_day_(prograde).png#/media/File:Sidereal_day_(prograde).png

Μήνας

Ο μήνας στα αρχαία χρόνια προσδιοριζόταν από τις διαφορετικές φάσεις του φεγγαριού, αυτό είχε ως αποτέλεσμα, για να συγχρονιστεί με το έτος, να εισάγονται επιπλέον μήνες στο ημερολόγιο. Ο μήνας με την υιοθέτηση του Γρηγοριανού ημερολογίου έχει αποσυνδεθεί από την άμεση σχέση του με το φεγγάρι, έχει μέγεθος από 28 έως 31 ημέρες και 12 μήνες αποτελούν ένα έτος. Ωστόσο σε άλλα ημερολόγια συναντάμε λιγότερους ή περισσότερους μήνες, με διαφορετικές διάρκειες.

Έτος

Το έτος ορίζεται ως ο χρόνος μία πλήρους περιστροφής της γης γύρω από τον ήλιο. Για την μέτρησή του χρησιμοποιείται το θερινό ή το χειμερινό ηλιοστάσιο καθώς και η θερινή ή η χειμερινή

ισημερία, ενώ σε άλλες περιπτώσεις το πέρασμα του ήλιου από τον ζωδιακό κύκλο ή μακρινά αστέρια ως σταθερά σημεία. Σήμερα σύμφωνα με το Γρηγοριανό ημερολόγιο το έτος έχει διάρκεια 365 ή 366 ημέρες, ή κατά μέσο όρο 365,2425 ημέρες. Στις μέρες μας το ηλιακό ή τροπικό έτος διαρκεί περίπου 365,24219878 ημέρες, ενώ το αστρικό περίπου 365,256363004 ημέρες.

Περίοδος (Era)

Οι περίοδοι δεν συνδέονται με κάποιο αστρονομικό φαινόμενο αλλά κυρίως με διάφορα γεωλογικά, θρησκευτικά, κοινωνικά ή πολιτικά γεγονότα και για αυτόν τον λόγο δεν έχουν κάποια σταθερή διάρκεια. Στο Γρηγοριανό ημερολόγιο υπάρχουν δύο περίοδοι, η περίοδος προ Χριστού και μετά Χριστόν. Σε άλλα ημερολόγια μπορεί να έχουμε περισσότερες περιόδους, όπως για παράδειγμα στο Ιαπωνικό ημερολόγιο που έχουμε νέα περίοδο κάθε φορά που υπάρχει νέος αυτοκράτορας.

3.2 Προτυποποίηση του Χρόνου

3.2.1 Ημερολόγιο - Δίσεκτο Έτος

Πριν το Ιουλιανό ημερολόγιο

Από την αρχαιότητα μέχρι και περίπου μισό αιώνα προ Χριστού τα ημερολόγια προσπαθούσαν να συνδυάσουν την περιοδική κίνηση της γης γύρω από τον ήλιο με τις φάσεις της σελήνης. Αυτό συχνά οδηγούσε στην προσθήκη ή αφαίρεση μηνών, καθώς και σε αυξομοιώσεις στο μέγεθός τους.

Το ημερολόγιο όπως το γνωρίζουμε σήμερα, πήρε μια πρώτη μορφή την εποχή της ρωμαϊκής αυτοκρατορίας βασιζόμενο σε κάποια ελληνικά ημερολόγια. Λίγο πριν το Ιουλιανό, το ρωμαϊκό ημερολόγιο αποτελούταν από 12 μήνες που συνολικά δημιουργούσαν ένα έτος 355 ημερών που ιδανικά θα εναλλασσόταν περιοδικά με ένα έτος με πρόσθετο μήνα. Το έτος με τον πρόσθετο μήνα συνολικά θα αποτελούταν από 377 ή 378 ημέρες.

Το παραπάνω ημερολόγιο κατά μέσο όρο θα ακολουθούσε αρκετά πιστά το ηλιακό έτος, ωστόσο στην πραγματικότητα ο πρόσθετος μήνας επηρεαζόταν από πολιτικά και όχι μόνο, γεγονότα και συμφέροντα, έτσι δεν είχε κάποια σταθερή περίοδο με αποτέλεσμα να δημιουργεί σύγχυση, συν το γεγονός ότι σε απομακρυσμένα μέρη της αυτοκρατορίας υπήρχε καθυστερημένη ενημέρωση για την προσθήκη του πρόσθετου μήνα.

Το Ιουλιανό ημερολόγιο

Ο Ιούλιος Καίσαρας το 46 π.Χ. για να λύσει αυτού του είδους τα προβλήματα αναθεώρησε το ρωμαϊκό ημερολόγιο και εισήγαγε το Ιουλιανό ημερολόγιο που ήρθε να αποσυνδέσει, την άμεση σχέση του έτους από το αστρονομικό φαινόμενο των φάσεων του φεγγαριού.

Το Ιουλιανό ημερολόγιο χωριζόταν σε 12 μήνες όπως τους ξέρουμε σήμερα με σύνολο 365 ημέρες, ενώ κάθε 4 χρόνια υπήρχε πρόσθεση μία ημέρας στον μήνα Φεβρουάριο (δίσεκτο έτος). Το ημερολόγιο αυτό έδινε κατά μέσο όρο 365,25 ημέρες το κάθε έτος, όπου η τιμή αυτή προσέγγιζε αρκετά τιμή του ηλιακού έτους.

Αν και το ηλιακό έτος ήταν κάποια λεπτά πιο σύντομο και η διαφορά αυτή ήταν γνωστή νωρίτερα από αστρονόμους της εποχής, δεν λήφθηκε υπόψιν, με αποτέλεσμα το Ιουλιανό ημερολόγιο να προηγείται κατά τρεις ημέρες του ηλιακού κάθε 4 αιώνες. Αυτό το πρόβλημα ήρθε να λύσει το Γρηγοριανό ημερολόγιο.

Το Γρηγοριανό ημερολόγιο

Το 1582 ο Πάπας Γρηγόριος ΙΓ’ αναθεώρησε το Ιουλιανό ημερολόγιο. Το αναθεωρημένο ημερολόγιο πήρε το όνομά του και πλέον χρησιμοποιείται ως καθιερωμένο ημερολόγιο διεθνώς. Η κύρια αλλαγή που έγινε ήταν στον υπολογισμό των δίσεκτων ετών, ενώ μέχρι τότε κάθε 4 αιώνες υπήρχαν

100 δίσεκτα έτη, με τον κανόνα ότι όποιο έτος διαιρείται με το 100 αλλά όχι με το 400 δεν είναι δίσεκτο πλέον κάθε 4 αιώνες θα υπήρχαν 97 δίσεκτα έτη.

Με αυτόν τον τρόπο ο μέσος όρος του έτους στο Γρηγοριανό ημερολόγιο είναι 365.2425 ημέρες που προσεγγίζει πολύ περισσότερο το ηλιακό έτος. Το σφάλμα του σε σχέση με το ηλιακό έτος είναι 1 ημέρα κάθε 3300 έτη, ενώ αν συμπεριληφθεί και η ακρίβεια των μετρήσεων των ισημεριών, τότε το σφάλμα γίνεται 1 ημέρα κάθε 7700 έτη. Παρότι από τότε έχουν προταθεί ημερολόγια με μεγαλύτερη ακρίβεια, αυτά δεν έχουν επισήμως υιοθετηθεί.

Η αποδοχή παγκοσμίως του γρηγοριανού ημερολογίου έγινε σταδιακά, στις περισσότερες περιπτώσεις οι λόγοι μη μετάβασης ήταν θρησκευτικοί, όπως για παράδειγμα στην Ελλάδα όπου η μετάβαση έγινε μόλις το 1923 και ήταν μία από τις τελευταίες χώρες που έκαναν την μετάβαση αυτή.

Σήμερα μεγάλη πλειοψηφία των χωρών χρησιμοποιεί το Γρηγοριανό ημερολόγιο ως το καθημερινό ημερολόγιο. Κάποιες χώρες δεν έχουν υιοθετήσει το ημερολόγιο, όπως η Σαουδική Αραβία, το Νεπάλ, η Αιθιοπία, το Ιράν και το Αφγανιστάν. Επίσης υπάρχουν χώρες που χρησιμοποιούν τροποποιημένη έκδοση του ημερολογίου όπως η Ταϊλάνδη, η Ιαπωνία, η Βόρειος Κορέα και η Ταϊβάν. Τέλος υπάρχουν πολλοί θρησκευτικοί οργανισμοί που δεν έχουν υιοθετήσει το Γρηγοριανό ημερολόγιο και χρησιμοποιούν άλλα εναλλακτικά.

3.2.2 Ώρα και Ζώνες Ώρας

Πριν τον μέσο χρόνο Γκρίνουιτς (GMT)

Η υποδιαίρεση και η μέτρηση της ώρας γινόταν στα αρχαία χρόνια μέσω των ηλιακών ρολογιών παρατηρώντας την θέση του ήλιου μέσω της σκιάς που δημιουργούσε και με την χρήση των κλεψύδρων όπου γινόταν η μέτρηση διαστημάτων του χρόνου με σχετική ακρίβεια.

Αργότερα κατασκευάστηκαν ρολόγια με μηχανισμό διαφυγής που εκμεταλλεύονταν την ροή του νερού και την βαρύτητα. Έγινε επίσης συνηθισμένο σε κεντρικά σημεία των πόλεων να υπάρχουν ρολόγια πύργοι τα οποία ειδοποιούσαν με ηχητικά σήματα όπως χτύπημα καμπάνας, για την ώρα. Πολλά από αυτά έδειχναν την ώρα και στις όψεις τους.

Μόλις στις αρχές του 14ου αιώνα κατασκευάστηκαν τα πρώτα μηχανικά ρολόγια και αποτέλεσαν τις βασικές συσκευές μέτρησης του χρόνου μέχρι το 1656 όπου κατασκευάστηκε για πρώτη φορά το ρολόι τύπου εκκρεμές.

Παράλληλα, με την εφεύρεση των ελατηρίων, άνοιξε ο δρόμος για τα κινητά ρολόγια και την εξάπλωσή τους σε όλο και περισσότερο κόσμο. Αυτό οδήγησε στην ανάγκη θέσπισης κοινής ώρας, αρχικά στο επίπεδο μίας πόλης και έπειτα στο επίπεδο περιοχών.

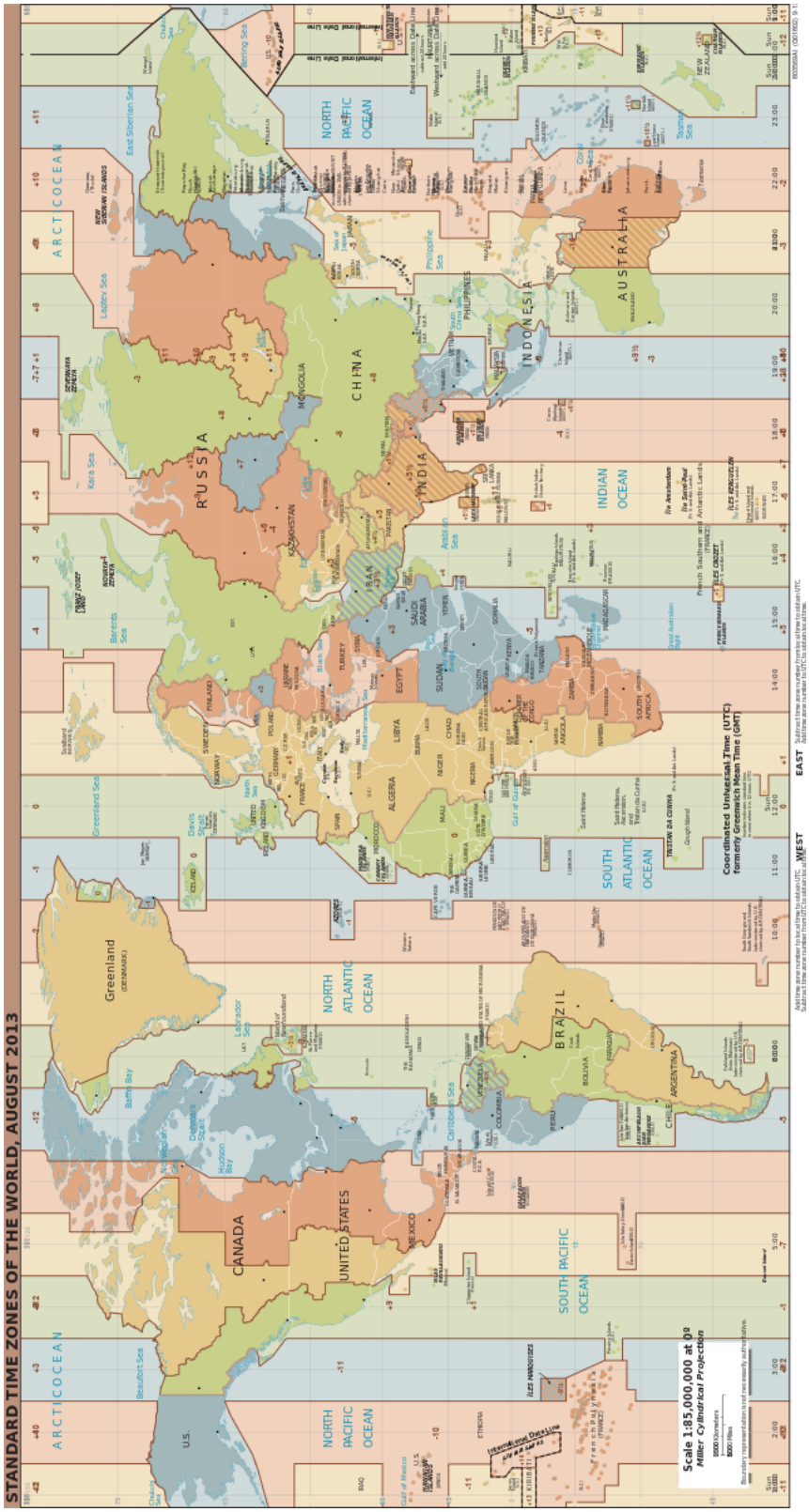
Μέσος Χρόνος Γκρίνουιτς (Greenwich Mean Time - GMT)

Το 1675 ιδρύθηκε το Βασιλικό Αστεροσκοπείο του Γκρίνουιτς όπου μέσα στους σκοπούς του ήταν να παρατηρεί αστρονομικά φαινόμενα, καθώς και να διατηρεί μία ώρα αναφοράς. Η ώρα αναφοράς χρησιμοποιήθηκε σε συνδυασμό με ρολόγια ναυσιπλοΐας, ώστε να μπορούν τα πλοία να υπολογίζουν το γεωγραφικό μήκος στο οποίο βρίσκονται. Αυτό αποτέλεσε σημαντικό εργαλείο και έδωσε ώθηση στην εξερεύνηση των θαλασσών.

Εκτός όμως από τα θαλάσσια ταξίδια, αργότερα με την εξέλιξη των σιδηροδρομικών γραμμών σε μεγάλες αποστάσεις, καθώς επίσης και με την εξέλιξη στις τηλεπικοινωνίες, υπήρξε η ανάγκη μία κοινής τοπικής ώρας. Έτσι σιγά σιγά υιοθετήθηκε η κοινή ώρα σε κοντινές περιοχές και αποτέλεσαν τον πρόδρομο των ζωνών ώρας.

Ο ορισμός των ζωνών ώρας καθώς και της Κοινής Παγκόσμιας Ώρας ήρθε από το διεθνές συνέδριο για του μεσημβρινούς το 1884 με την θέσπιση κοινών μεσημβρινών και ως ώρα αναφοράς τον Μέσο Χρόνο Γκρίνουιτς, κυρίως για αστρονομικές μετρήσεις και όχι τόσο για καθημερινή χρήση. Σταδιακά τις επόμενες δεκαετίες οι ζώνες ώρας, καθώς και η διαφορά τους από την παγκόσμια κοινή ώρα υιοθετήθηκαν από τις διάφορες χώρες.

Σχήμα 3.2: Ζώνες Ώρας (Αύγουστος 2015)



Source: ”Standard World Time Zones” by TimeZonesBoy - Own work. Licensed under CC BY-SA 4.0 via Commons - https://commons.wikimedia.org/wiki/File:Standard_World_Time_Zones.png#/media/File:Standard_World_Time_Zones.png

Αρχικά η διαφορά της τοπικής από την κοινή ώρα δεν ήταν πάντα σε ακέραιες ώρες αν και στην πορεία ομαλοποιήθηκε η κατάσταση αυτή. Έτσι οι διαφορές των τοπικών ωρών με την κοινή ώρα στις περισσότερες χώρες πλέον είναι στην πλειοψηφία τους πολλαπλάσια της ώρας με κάποιες εξαιρέσεις, όπου χρησιμοποιείται η μισή ώρα, και ακόμα πιο λίγες εξαιρέσεις όπου χρησιμοποιείται το τέταρτο της ώρας, όπως για παράδειγμα στο Νεπάλ.

Ζώνες Ώρας (TimeZones)

Ο αυστηρός ορισμός της ζώνης ώρας συνδέει την τοπική ώρα ενός εύρους γεωγραφικών μηκών με την κοινή ώρα μέσω μίας σταθεράς πολλαπλάσιας της μίας ώρας. Αυτός ο ορισμός δεν ακολουθείται πλήρως από τις διάφορες χώρες καθώς προκύπτουν γεωγραφικά, πολιτικά και κοινωνικά ζητήματα.

Πολλές χώρες αλλάζουν περιοδικά την τοπική τους ώρα κατά τους θερινούς μήνες (θερινή ώρα) και επιστρέφουν πίσω σε αυτήν τους χειμερινούς (χειμερινή ώρα). Όμως υπάρχουν και χώρες που κάνουν αυτές τις αλλαγές εκτάκτως ή πειραματικά. Αν προσθέσουμε και τις χώρες που αλλάζουν της τοπικές τους ώρες για άλλους λόγους τότε η παρακολούθηση των τοπικών ωρών γίνεται ένα σύνθετο πρόβλημα.

Αυτό το πρόβλημα προσπάθησε να λύσει με μία βάση δεδομένων Ο Άρθουρ Ντέιβιντ Όλσον, μία βάση δεδομένων όπου θα κατέγραφε όλες τις αλλαγές των τοπικών ωρών παγκοσμίως. Η βάση αυτή είναι κοινό κτήμα (public domain) και πλέον συντηρείται από τον Πωλ Έγκερτ και τον Φορέα του Διαδικτύου για την Εκχώρηση Ονομάτων και Αριθμών (ICANN) με συνεισφορές από εθελοντές και διατίθεται μέσω της Αρχής Εκχώρησης Αριθμών Διαδικτύου (IANA).

Η ακρίβεια της βάσης για τις τοπικές ώρες παγκοσμίως ξεκινά από το 1970 και φτάνει μέχρι σήμερα όπου ανανεώνεται περιοδικά. Για πριν το 1970 υπάρχουν μεμονωμένα στοιχεία για κάποιες περιοχές και χώρες. Η βάση αυτή είναι η πιο διαδεδομένη και χρησιμοποιείται ευρύτατα σε διάφορους τομείς παρότι υπάρχουν και συντηρούνται ανάλογες βάσεις και από άλλους οργανισμούς όπως η Microsoft και η Διεθνής Ένωση Αερομεταφορών (IATA).

3.2.3 Κοινή Παγκόσμια Ώρα

Κοινή Παγκόσμια Ώρα - Ομοιόμορφη Ώρα (Universal Time - Uniform Time)

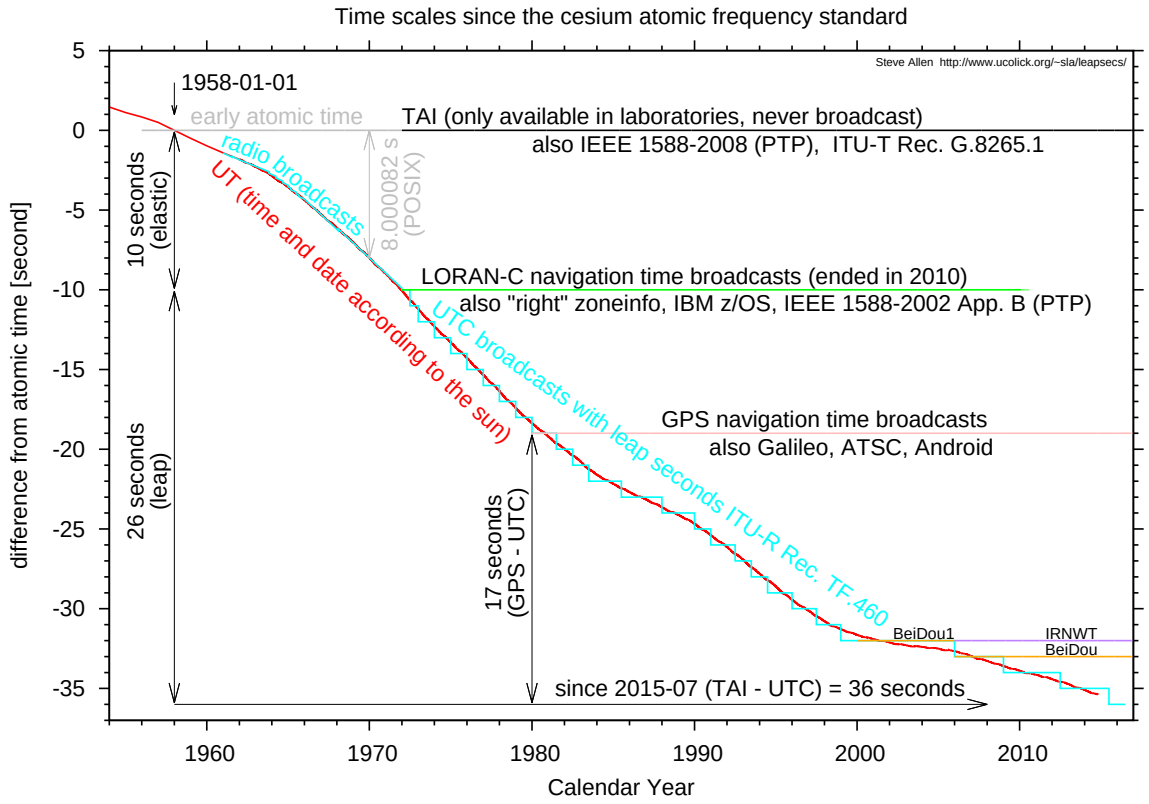
Με το συνέδριο του 1884 ορίστηκε η Κοινή Παγκόσμια Ώρα (Universal Time - UT) με αναφορά τον Μέσο Χρόνο Γκρίνουιτς και βασιζόμενη στις αστρονομικές παρατηρήσεις της περιστροφής της γης. Ωστόσο η περιστροφή της γης εμφανίζει ανωμαλίες καθώς και μία σταθερή επιβράδυνση, για αυτό τον λόγο αναζητήθηκε μία ώρα η οποία θα ήταν ομοιόμορφη και δεν θα εξαρτώταν από την περιστροφή της γης όπως η αστρική(sidereal time) ή η κοινή παγκόσμια ώρα(UT). Αυτή επίσης θα όριζε και την μονάδα του δευτερολέπτου.

Η πρώτη προσπάθεια να υπάρξει τέτοια ώρα ήρθε το 1952 με την Αστρονομική Ώρα (Ephemeris Time) βασίστηκε στην κίνηση της Γης γύρω από τον ήλιο και την Νευτώνια φυσική. Το 1984 αντικαταστάθηκε από τον συνδυασμό της Επίγειας Δυναμικής Ώρας (Terestrial Dynamic Time) και της Βαρυκεντρικής Δυναμικής Ώρας (Barycentric Dynamical Time), όπου με την σειρά τους αντικαταστάθηκαν από την Επίγεια Ώρα (Terestrial Time) που βασιζόταν πλέον στο ορισμένο από ατομικά ρολόγια δευτερόλεπτο και στην γενική θεωρία της σχετικότητας.

Συγχρονισμένη Παγκόσμια Ώρα - Διεθνής Ατομική Ώρα (Coordinated Universal Time - International Atomic Time)

Παράλληλα με την Ομοιόμορφη Ώρα άρχισαν να αναπτύσσονται τα ατομικά ρολόγια όπου ουσιαστικά αναθεωρούσαν και όριζαν καλύτερα την ιδέα αυτής. Έτσι ορίστηκε το 1958 η Διεθνής Ατομική Ώρα (International Atomic Time - TAI) στην αρχική μορφή της και με κάποιες διορθώσεις, οδήγησε στην αναθεώρηση του ορισμού του δευτερολέπτου το 1967 και επηρέασε τον ορισμό της Επίγειας Ώρας (Terestrial Time).

Σχήμα 3.3: Κλίμακες ώρας από τον ορισμό του δευτερολέπτου (1967)



Source: http://www.ucolick.org/ sla/leapsecs/amsci.html

Η Ατομική Ώρα επηρέασε και την Κοινή Παγκόσμια Ώρα καθώς πλέον υπήρχε η ανάγκη από την μια η ώρα να συμβαδίζει με την περιστροφή της Γης και από την άλλη τα δευτερόλεπτα να μην έχουν διαφορετική διάρκεια μεταξύ τους. Οι δύο αυτοί ορισμοί είναι συγκρουόμενοι αλλά η μέση λύση δόθηκε με τον ορισμό το 1960 της Συγχρονισμένης Παγκόσμιας Ώρας και την συμπλήρωσή της το 1972 με τα άλματα δευτερολέπτου (leap seconds).

Η Συγχρονισμένη Παγκόσμια Ώρα ικανοποιεί την σταθερή διάρκεια των δευτερολέπτων αφού βρίσκεται σε σταθερή διαφορά δευτερολέπτων από την Διεθνή Ατομική Ώρα αλλά και συμβαδίζει με την περιστροφή της Γης και την Κοινή Παγκόσμια Ώρα με τον μηχανισμό του άλματος δευτερολέπτου.

Άλμα δευτερολέπτου (Leap Second)

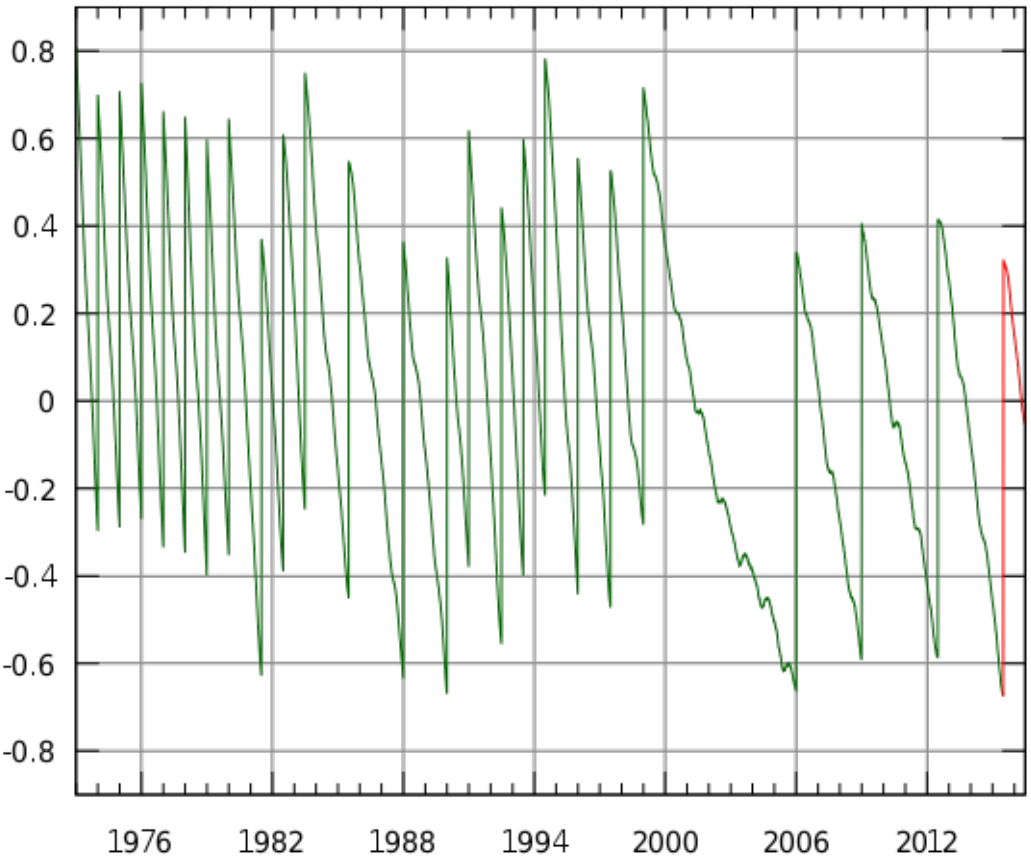
Σχεδόν κάθε φορά που η διαφορά της Συγχρονισμένης από την Κοινή Παγκόσμια Ώρα προσεγγίζει τα 0.6 δευτερόλεπτα προστίθεται ή αφαιρείται ένα δευτερόλεπτο (leap second) έτσι ώστε η διαφορά να μην ξεπεράσει ποτέ την τιμή των 0.9 δευτερολέπτων.

Από την έναρξη του μηχανισμού για τα άλματα δευτερολέπτου το 1972 δεν έχει γίνει αφαίρεση κάποιου δευτερολέπτου, ενώ συνολικά έχουν προστεθεί 26 δευτερόλεπτα μέχρι και σήμερα (Οκτώβρης 2015). Αυτό σημαίνει ότι με την αρχική διαφορά 10 δευτερολέπτων της Συγχρονισμένης Παγκόσμιας από την Διεθνή Ατομική Ώρα, σήμερα η διαφορά βρίσκεται στα 36 δευτερόλεπτα (TAI-UTC).

Το πρότυπο του συστήματος του άλματος δευτερολέπτου ορίζει ότι το άλμα μπορεί να γίνει στο τέλος οποιουδήποτε μήνα με πρώτη προτίμηση τον Ιούνιο ή τον Δεκέμβριο και με δεύτερη προτίμηση τον Μάρτιο ή τον Σεπτέμβριο. Μέχρι σήμερα έχουν προστεθεί μόνο τους μήνες Ιούνιο και Δεκέμβριο.

Υπεύθυνη για την ανακοίνωση των αλμάτων είναι η Διεθνής Υπηρεσία για την Περιστροφή της Γης και Συστημάτων Αναφοράς(IERS). Η IERS ανακοινώνει κάθε έξι μήνες αν θα γίνει ή όχι άλμα καθώς δεν μπορεί να γίνει πιο μακρινή πρόβλεψη των μεταβολών της περιστροφής της γης.

Σχήμα 3.4: Διαφορά Κοινής και Συγχρονισμένης Παγκόσμιας Ώρας
(οι κάθετες γραμμές αντιστοιχούν σε άλματα δευτερολέπτων)



Source: ”Leapsecond.ut1-utc” by Tomia (talk · contribs)Petr Kadlec (talk · contribs)Gordon P. Hemsley (talk · contribs)RP88 (talk · contribs) - Own work using: http://maia.usno.navy.mil/ser7/finals.all. Licensed under Public Domain via Commons - https://commons.wikimedia.org/wiki/File:Leapsecond.ut1-utc.svg#/media/File:Leapsecond.ut1-utc.svg

3.2.4 Πρότυπο ISO 8601

Ο διεθνής οργανισμός προτυποποίησης το 1988 εξέδωσε το πρότυπο ISO 8601 στο οποίο περιέγραψε τους τρόπους με τους οποίους θα παρουσιάζονται οι ημερομηνίες και οι ώρες. Το πρόβλημα που ήρθε να καλύψει είναι η μη καλά ορισμένη, διφορούμενη καθώς και διαφορετική αναπαράσταση της ημερομηνίας και της ώρας σε διάφορες χώρες.

Το πρότυπο έχει ως βάση του και εφαρμόζεται στο Γρηγοριανό ημερολόγιο και την 24ωρη αναπαράσταση της ώρας. Οι τιμές των μεγεθών ακολουθούν την ταξινομημένη σειρά από τη μεγαλύτερη προς την μικρότερη τιμή: Έτος, Μήνας(ή Εβδομάδα), Ημέρα, Ώρα, Λεπτό, Δευτερόλεπτο και κλάσμα του Δευτερολέπτου.

Περιγράφει επίσης την αναπαράσταση διάρκειας και επαναλαμβανόμενων ή μη χρονικών διαστημάτων, καθώς και την αναπαράσταση ζωνών ώρας με την μορφή της διαφοράς τους σε ώρες και λεπτά από την Συγχρονισμένη Παγκόσμια Ώρα.

Κεφάλαιο 4

Βιβλιοθήκη Ημερομηνίας και Ώρας

4.1 Σκοπός

Ο σκοπός της δημιουργίας της βιβλιοθήκης αυτής είναι να αποτελέσει ένα εκτενές μοντέλο που θα υλοποιεί τις βασικές έννοιες που σχετίζονται με τον χρόνο, την λειτουργικότητα τους, καθώς και την μεταξύ τους αλληλεπίδραση και θα εξυπηρετεί τις ανάγκες της πλειονότητας των εφαρμογών που ασχολούνται με τον χρόνο.

Στην πορεία αναζήτησης υλοποιήσεων και προδιαγραφών παρόμοιων βιβλιοθηκών σε άλλες γλώσσες και υπολογιστικά συστήματα, διαπιστώθηκε ότι ο παραπάνω σκοπός καλύπτεται σε σημαντικό βαθμό από τις προδιαγραφές JSR-000310. Έτσι η υλοποίηση της βιβλιοθήκης επηρεάστηκε και διαμορφώθηκε από τις προδιαγραφές αυτές.

4.2 Αρχές Σχεδίασης

Μία από της βασικές αρχές σχεδίασης ήταν η μη μεταβλητότητα (immutability) των οντοτήτων της βιβλιοθήκης η οποία θα επέτρεπε στην βιβλιοθήκη να είναι ασφαλής σε πολυπύρηνο περιβάλλον. Αυτό ήταν κάτι που μπορούσε εύκολα να καλυφθεί καθώς η γλώσσα προγραμματισμού Rust προσφέρει αυτήν την ασφάλεια εγγενώς.

Βασική αρχή αποτέλεσε επίσης η εκφραστικότητα της βιβλιοθήκης, και με την έννοια ότι η χρήση της θα εξασφάλιζε αναγνώσιμο κώδικα κοντά στην πραγματική γλώσσα, αλλά και με την έννοια της πλούσιας λειτουργικότητας.

Επιπλέον οι έννοιες, οι μέθοδοι και οι συναρτήσεις θα έπρεπε να είναι απλές, κατανοητές και καλά ορισμένες. Ταυτόχρονα, η βιβλιοθήκη θα έπρεπε να δίνει και την δυνατότητα επέκτασης της στον προγραμματιστή, ώστε να μπορεί να υλοποιήσει έννοιες του χρόνου που δεν ήταν δυνατό να προβλεφθούν καθώς είτε δεν ήταν γνωστές είτε αποτελούν ειδικές περιπτώσεις.

4.3 Απαιτήσεις

Πέρα από τις αρχές σχεδίασης έπρεπε να καλυφθούν και τρεις απαιτήσεις που πηγάζουν από την έννοια του χρόνου έτσι όπως την συναντάμε και την χρησιμοποιούμε σήμερα. Η πρώτη απαίτηση ήταν ο χειρισμός του συνεχόμενου χρόνου, δηλαδή την περιγραφή του χρόνου ως ένα συνεχόμενα αυξανόμενο αριθμό. Αυτήν είναι η αντίληψη του χρόνου από την σκοπιά μίας μηχανής.

Η δεύτερη απαίτηση ήταν ο χειρισμός του ανθρώπινου χρόνου, δηλαδή την περιγραφή του χρόνου με διακριτά πεδία όπως αυτό του έτους, του μήνα, της μέρας, της ώρα, του λεπτού, του δευτερολέπτου κλπ. Με λίγα λόγια όπως τον αντιλαμβάνεται και τον χρησιμοποιεί ο άνθρωπος. Τέλος η τρίτη απαίτηση ήταν η μετατροπή του χρόνου από την συνεχόμενη στη ανθρώπινη μορφή και αντίστροφα, καθώς και η ορισμένη συνάφεια των δύο αυτών μορφών.

4.4 Δομή

Η δομή της βιβλιοθήκης έχει την παρακάτω μορφή:

- **Κυρίως σώμα:** περιλαμβάνει τον ορισμό όλων των βασικών εννοιών της ημερομηνίας και της ώρας, που αναλύονται παρακάτω, καθώς και την σχετική λειτουργικότητα που τις ακολουθεί. Οι έννοιες αυτές βασίζονται και ακολουθούν το πρότυπο ISO 8601 με ελάχιστες παρεκκλίσεις.
- **Chrono:** αποτελείται από τις γενικές έννοιες της χρονολογίας, της περιόδου και της ημερομηνίας όπως αυτές παρουσιάζονται στην έννοια του ημερολογίου, καθώς και την λειτουργικότητά τους. Το σύνολο αυτών των εννοιών και των λειτουργιών τους οφείλει να υλοποιήσει ο προγραμματιστής, όπου αυτές έχουν νόημα, για την αναπαράσταση ενός άλλου ημερολογίου.
- **Temporal:** περιέχει τις έννοιες και την λειτουργικότητα που πρέπει να υλοποιηθούν για να ορισθούν τα αντικείμενα της ημερομηνίας και της ώρας. Επίσης περιλαμβάνει την λειτουργικότητα και τον ορισμό των ρυθμιστών(adjusters) και των ερωτημάτων(queries) πάνω σε αυτά τα αντικείμενα.
- **Zone:** περιέχει τις έννοιες και την λειτουργικότητα που σχετίζονται με τις ζώνες ώρας, όπως την διαχείριση κανόνων και αναγνωριστικών των ζωνών ώρας.

4.5 Βασικές Έννοιες

4.5.1 Συνεχόμενος Χρόνος (Χρόνος Μηχανής)

Γραμμή του χρόνου (Time-Line)

Πρόκειται για την έννοια του χρόνου όπου αναπαρίσταται από μία κατευθυνόμενη γραμμή από το παρελθόν προς το μέλλον. Αυτός είναι συνήθως ο τρόπος με τον οποίο ο άνθρωπος απεικονίζει η/και αναπαριστά την έννοια του χρόνου.

Χρονική Στιγμή (Instant)

Η χρονική στιγμή είναι ένα συγκεκριμένο σημείο πάνω στην γραμμή του χρόνου και αποτελεί την χρονοσήμανση (timestamp) ενός γεγονότος με ακρίβεια. Την έννοια αυτή την υλοποιήσαμε κάνοντας χρήση μίας απλής δομής με δύο πεδία:

- **Δευτερόλεπτο:** Τα δευτερόλεπτα της χρονικής στιγμής
- **Κλάσμα Δευτερολέπτου:** Το κλασματικό μέρος της χρονικής στιγμής μετρούμενο σε νανοδευτερόλεπτα με εύρος [0-999.999.999]

Με την κατασκευή αυτή μπορούμε να περιγράψουμε οποιαδήποτε χρονική στιγμή εντός της ηλικίας του σύμπαντος. Ως χρονική στιγμή μηδέν ορίζουμε την ημερομηνία 1970-01-01.

Διάρκεια (Duration)

Η διάρκεια δεν αποτελεί μέρος της γραμμής του χρόνου καθώς είναι ποσότητα. Η ποσότητα μεταξύ δύο χρονικών στιγμών. Όπως και η χρονική στιγμή αποτελείται από μία απλή δομή με δύο πεδία:

- **Δευτερόλεπτο:** Η διάρκεια σε δευτερόλεπτα
- **Κλάσμα Δευτερολέπτου:** Το κλασματικό μέρος της διάρκειας μετρούμενο σε νανοδευτερόλεπτα με εύρος [0-999.999.999]

4.5.2 Ανθρώπινος Χρόνος

Τοπική ημερομηνία (Local date)

Πρόκειται για την ημερομηνία χωρίς κάποια αναφορά σε ώρα ή ζώνη ώρας. Αποτελείται από μία απλή δομή με τρία πεδία:

- **Έτος:** Το έτος της ημερομηνίας
- **Μήνας:** Ο μήνας τους έτους σε αριθμητική μορφή με εύρος [1-12]
- **Ημέρα:** Η ημέρα του μήνα σε αριθμητική μορφή με εύρος [1-31]

Παραδείγματα εφαρμογών είναι η ημερομηνία γέννησης ή ημερομηνία μιας γιορτής.

Τοπική ώρα (Local time)

Περιγράφει την ώρα χωρίς κάποια αναφορά σε ημερομηνία ή ζώνη ώρας. Αποτελείται από μία απλή δομή με τέσσερα πεδία:

- **Ώρα:** Η τιμή της ώρας (hour) της ώρας (time) με εύρος [0-23]
- **Λεπτό:** Το λεπτό της ώρας με εύρος [0-59]
- **Δευτερόλεπτο:** Το δευτερόλεπτο του λεπτού με εύρος [0-59]
- **Κλάσμα δευτερολέπτου:** Το κλασματικό μέρος του δευτερολέπτου μετρούμενο σε νανοδευτερόλεπτα με εύρος [0-999.999.999]

Παραδείγματα χρήσης είναι η ώρα έναρξης του ωραρίου των καταστημάτων ή η ώρα όπως περιγράφεται από ένα ξυπνητήρι.

Τοπική ημερομηνία-ώρα (Local date-time)

Αποτελεί τον συνδυασμό ημερομηνίας και ώρας χωρίς κάποια αναφορά σε ζώνη ώρας. Αποτελείται από μία απλή δομή με δύο πεδία:

- **Τοπική ημερομηνία:** Η τοπική ημερομηνία του συνδυασμού
- **Τοπική ώρα:** Η τοπική ώρα του συνδυασμού

Μπορεί να χρησιμοποιηθεί για την ημερομηνία-ώρα απογείωσης μιας πτήσης ή την ημερομηνία-ώρα μιας συναυλίας.

Έτος (Year)

Πρόκειται για την καταγραφή ενός έτους χωρίς κάποια αναφορά σε άλλο μέγεθος. Αποτελείται από μία απλή δομή με ένα πεδίο που περιγράφει αριθμητικά το έτος.

Μήνας (Month)

Περιγράφει ένα μήνα χωρίς κάποια αναφορά σε άλλο μέγεθος. Αποτελείται από μία απαρίθμηση που απαριθμεί ονομαστικά τους 12 μήνες.

Έτος-Μήνας (Year-Month)

Αποτελεί τον συνδυασμό ενός έτους και ενός μήνα και υλοποιείται με μία απλή δομή με δύο πεδία:

- **Έτος:** Το έτος της ημερομηνίας
- **Μήνας:** Ο μήνας τους έτους σε αριθμητική μορφή με εύρος [1-12]

Μήνας-Ημέρα (Month-Day)

Είναι ο συνδυασμός ενός μήνα και μίας ημέρας και υλοποιείται με μία απλή δομή με δύο πεδία:

- **Μήνας:** Ο μήνας ενός μη ορισμένου έτους σε αριθμητική μορφή με εύρος [1-12]
- **Ημέρα:** Η ημέρα του μήνα σε αριθμητική μορφή με εύρος [1-31]

Ημέρα της εβδομάδας (Day of week)

Πρόκειται για την καταγραφή μία ημέρας της εβδομάδας χωρίς κάποια αναφορά σε άλλο μέγεθος. Αποτελείται από μία απαρίθμηση που απαριθμεί ονομαστικά τις 7 ημέρες της εβδομάδας.

Μετατόπιση ζώνης (Zone offset)

Πρόκειται για την μετατόπιση σε ώρες και λεπτά από την Συγχρονισμένη Παγκόσμια Ώρα. Μπορεί να έχει θετική ή αρνητική τιμή και τα όριά της είναι -18:00 έως +18:00. Αποτελείται από μία δομή με ένα πεδίο που περιγράφει την μετατόπιση σε δευτερόλεπτα με εύρος [-64800, 64800].

Μετατοπισμένη ημερομηνία-ώρα (Offset date-time)

Αποτελεί τον συνδυασμό ημερομηνίας και ώρας με μετατόπιση ζώνης. Αποτελείται από μία απλή δομή με δύο πεδία

- **Τοπική ημερομηνία-ώρα:** Η τοπική ημερομηνία-ώρα του συνδυασμού
- **Μετατόπιση ζώνης:** Η μετατόπιση ζώνης του συνδυασμού

Πρόκειται για παρόμοια έννοια με την χρονική στιγμή αλλά παρέχοντας επιπλέον πληροφορίες. Χρησιμοποιείται για χρονοσήμανση με μετατόπιση ζώνης ενός γεγονότος.

Μετατοπισμένη ώρα (Offset time)

Περιγράφει τον συνδυασμό τοπικής ώρας και μετατόπισης ζώνης και αποτελείται από δύο πεδία:

- **Τοπική ώρα:** Η τοπική ώρα του συνδυασμού
- **Μετατόπιση ζώνης:** Η μετατόπιση ζώνης του συνδυασμού

Κανόνες μετατόπισης (Zone Rules)

Πρόκειται για το σύνολο των κανόνων μετάβασης μιας ζώνης ώρας που περιγράφουν τις διάφορες αλλαγές της ώρας που έχουν συμβεί σε αυτήν την ζώνη ώρας.

Αναγνωριστικό Ζώνης (Zone Id)

Περιγράφει ένα σταθερό αναγνωριστικό ζώνης μίας τοποθεσίας/κυβέρνησης.

Ημερομηνία-Ώρα με ζώνη ώρας (Zoned date-time)

Αποτελεί τον συνδυασμό ημερομηνίας και ώρας με μία ζώνη ώρας. Αποτελείται από μία απλή δομή με τρία πεδία:

- **Τοπική ημερομηνία-ώρα:** Η τοπική ημερομηνία-ώρα του συνδυασμού
- **Αναγνωριστικό ζώνης:** Το αναγνωριστικό ζώνης του συνδυασμού
- **Μετατόπιση ζώνης:** Η μετατόπιση ζώνης του συνδυασμού

Περίοδος (Period)

Πρόκειται για την ποσότητα μεταξύ δύο χρονικών στιγμών που περιγράφεται με έννοιες της ημερομηνίας. Υλοποιείται με μία απλή δομή με τρία πεδία:

- **Έτη:** Τα έτη της διάρκειας της περιόδου
- **Μήνες:** Οι μήνες της διάρκειας της περιόδου
- **Ημέρες:** Οι ημέρες της διάρκειας της περιόδου

Τρέχουσα ώρα και ρολόι (clock)

Το ρολόι είναι η δομή που μας δίνει την τρέχουσα ώρα και επηρεάζεται από τις ζώνες ώρες. Η χρήση του δεν είναι απαραίτητη αλλά δίνει στον προγραμματιστή την δυνατότητα να ορίσει την τρέχουσα ώρα σε διαφορετική τιμή από αυτή του υπολογιστικού συστήματος που εκτελεί το πρόγραμμά του.

4.6 Λειτουργίες

4.6.1 Βασικές Μέθοδοι και Συναρτήσεις

Οι έννοιες που παρουσιάστηκαν υλοποιούν κάποιες βασικές μεθόδους και συναρτήσεις που έχουν κοινή σημασία τηρουμένων των αναλογιών:

- **`now`:** Οι συναρτήσεις που επιστρέφουν το αντικείμενο συμπληρώνοντας τα πεδία του με τις τιμές της τρέχουσας ημερομηνίας-ώρας.
- **`from`:** Οι συναρτήσεις που επιστρέφουν το αντικείμενο εξάγοντας το από ένα άλλο αντικείμενο.
- **`of`:** Οι συναρτήσεις που επιστρέφουν το αντικείμενο συμπληρώνοντας τα πεδία του με τις τιμές που δόθηκαν στις παραμέτρους της συνάρτησης.
- **get:** Οι μέθοδοι που επιστρέφουν την τιμή κάποιου πεδίου του αντικειμένου.
- **`is`:** Οι μέθοδοι που επιστρέφουν δυαδική τιμή για το αν το αντικείμενο ανήκει σε μία συγκεκριμένη κατηγορία.
- **`with`:** Οι μέθοδοι που επιστρέφουν αντίγραφο του αντικειμένου με αλλαγμένες κάποιες τιμές του οι οποίες δίνονται ως παράμετροι στην μέθοδο.
- **`plus`:** Οι μέθοδοι που επιστρέφουν αντίγραφο του αντικειμένου έχοντας προσθέσει την ποσότητα που δίνεται ως παράμετρος.
- **`minus`:** Οι μέθοδοι που επιστρέφουν αντίγραφο του αντικειμένου έχοντας αφαιρέσει την ποσότητα που δίνεται ως παράμετρος.

- **`to`:** Οι μέθοδοι που μετατρέπουν το αντικείμενο σε ένα άλλο χρησιμοποιώντας τις τιμές των πεδίων του.
- **`at`:** Οι μέθοδοι που επιστρέφουν το αντικείμενο συνδυασμένο με ένα άλλο αντικείμενο.

4.6.2 Ρυθμιστές (Adjusters)

Πρόκειται για ανώνυμες συναρτήσεις που μπορεί να κατασκευάσει ο προγραμματιστής και χρησιμεύουν στην ρύθμιση του τρέχοντος αντικειμένου μέσω πράξεων ημερομηνίας και ώρας. Η βιβλιοθήκη υλοποιεί κάποιους προεπιλεγμένους ρυθμιστές:

- **`next(day_of_week)`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην επόμενη ημέρα της εβδομάδας.
- **`next_or_same(day_of_week)`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην επόμενη ημέρα της εβδομάδας ή στην ίδια μέρα αν το αντικείμενο βρίσκεται σε αυτή.
- **`previous(day_of_week)`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην προηγούμενη ημέρα της εβδομάδας.
- **`previous_or_same(day_of_week)`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην προηγούμενη ημέρα της εβδομάδας ή στην ίδια μέρα αν το αντικείμενο βρίσκεται σε αυτή.
- **`first_in_month(day_of_week)`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην πρώτη ημέρα της εβδομάδας του μήνα.
- **`last_in_month(day_of_week)`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην τελευταία ημέρα της εβδομάδας του μήνα.
- **`first_day_of_month()`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην πρώτη ημέρα του μήνα.
- **`last_day_of_month()`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην τελευταία ημέρα του μήνα.
- **`first_day_of_year()`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην πρώτη ημέρα του έτους.
- **`last_day_of_year()`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην τελευταία ημέρα του έτους.
- **`first_day_of_next_month()`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην πρώτη ημέρα του επόμενου μήνα.
- **`first_day_of_next_year()`:** Επιστρέφει το αντικείμενο ρυθμισμένο στην πρώτη ημέρα του επόμενου έτους.

4.6.3 Ερωτήματα (Queries)

Πρόκειται για ανώνυμες συναρτήσεις που μπορεί να κατασκευάσει ο προγραμματιστής και χρησιμεύουν στην εξαγωγή πληροφοριών από το τρέχον αντικείμενο, αν υπάρχουν, μέσω πράξεων ημερομηνίας και ώρας. Η βιβλιοθήκη υλοποιεί κάποια προεπιλεγμένα ερωτήματα:

- **zone_id:** Επιστρέφει το αναγνωριστικό της ζώνης ώρας του αντικειμένου.
- **chronology:** Επιστρέφει την χρονολογία του αντικειμένου.
- **precision:** Επιστρέφει την ακρίβεια μέτρησης του αντικειμένου.
- **zone:** Επιστρέφει το αναγνωριστικό της ζώνης ώρας ή την μετατόπιση ζώνης του αντικειμένου.

- **offset:** Επιστρέφει την μετατόπιση της ζώνης του αντικειμένου.
- **local_date:** Επιστρέφει την τοπική ημερομηνία του αντικειμένου.
- **local_time:** Επιστρέφει την τοπική ώρα του αντικειμένου.

4.7 Ζητήματα

4.7.1 Ζώνες Ώρας (Time Zones)

Οι ζώνες ώρας στην θεωρία ορίζονται επακριβώς, όμως στην πράξη κάθε τοποθεσία/κυβέρνηση επιλέγει να ορίσει την δική της ζώνη ώρας, πολλές φορές με μετατοπίσεις από την μία ζώνη ώρας σε μια άλλη. Αυτό δημιουργεί ένα σύνθετο πρόβλημα που θέλει ειδικό χειρισμό.

Η αλλαγή του θεωρητικού ορισμού σε έναν ορισμό που θα περιγράφει την πρακτική εφαρμογή είναι αναγκαία. Για αυτό τον λόγο οι ζώνες ώρας ορίζονται με τον συνδυασμό ενός ονόματος και κάποιων κανόνων που περιγράφουν τις μεταβάσεις της ώρας. Γνωρίζοντας του κανόνες αυτούς μπορούμε να περιγράψουμε με ακρίβεια την ώρα.

Καταγραφή ονομάτων και κανόνων γίνεται τα τελευταία χρόνια από διάφορες οντότητες παγκοσμίως. Μία από τις πιο διαδεδομένες βάσεις δεδομένων είναι η Βάση Δεδομένων Ζωνών Ώρας (Timezone Database - TZDB) η οποία χρησιμοποιείται από αρκετά υπολογιστικά συστήματα καθώς αποτελεί και μία από τις πιο έγκυρες παγκοσμίως πηγές.

Για τους παραπάνω λόγους επιλέχθηκε να είναι η προεπιλεγμένη πηγή της βιβλιοθήκης όσον αφορά τις ζώνες ώρας. Ωστόσο δίνεται η δυνατότητα στον προγραμματιστή που θα χρησιμοποιήσει την βιβλιοθήκη να ορίσει ή να κάνει χρήση άλλων κανόνων με μόνο περιορισμό στο αναγνωριστικό (id) των ζωνών που θα ορίσει.

Τεχνικά η υλοποίηση έγινε με την χρήση της βιβλιοθήκης lazystatic η οποία προσφέρει την δυνατότητα αρχικοποίησης δυναμικών δομών, κάτι που λείπει από την γλώσσα Rust. Με αυτόν τον τρόπο την πρώτη φορά και μόνο που γίνεται χρήση κάποιας ζώνης ή κάποιου κανόνα ζώνης γίνεται η αρχικοποίηση κάποιων χαρτών κατακερματισμού και πινάκων με τα δεδομένα της βάσης δεδομένων και έπειτα μπορούν να χρησιμοποιηθούν χωρίς κάποιο άλλο κόστος αρχικοποίησης.

4.7.2 Κενά-Επικαλύψεις Ζώνης (Zone Gaps-Overlaps)

Κατά την εφαρμογή των κανόνων που ισχύουν για μία ζώνη ώρας όταν αυτοί αφορούν μετάβαση από την θερινή στην χειμερινή ώρα ή το αντίστροφο δημιουργούνται κενά και επικαλύψεις αντίστοιχα. Έτσι δημιουργείται το πρόβλημα ότι κάποιες ώρες είτε δεν υπάρχουν (κενό) είτε μπορούν να αντιστοιχούν σε δύο διαφορετικές ώρες (επικάλυψη).

Σε αυτό το ζήτημα μπορούν να υπάρξουν διάφορες λύσεις, μία από αυτές είναι η δημιουργία σφαλμάτων όταν συναντιόνται τέτοιες ημερομηνίες. Ωστόσο μία τέτοια λύση δεν είναι βολική για τον προγραμματιστή καθώς οι επικαλύψεις και τα κενά δεν είναι κάτι που προβλέπει συχνά και αυτό έχει ως αποτέλεσμα να προκύπτουν σφάλματα κατά την εκτέλεση των προγραμμάτων του και πολλές φορές στην διαδικασία παραγωγής.

Η παραπάνω συμπεριφορά δεν είναι επιθυμητή για τον μέσο προγραμματιστή έτσι επιλέχθηκε μία άλλη λύση, στην οποία σε περιπτώσεις κενών ή επικαλύψεων γίνεται ως προεπιλογή χρήση της θερινής ώρας όταν αυτό είναι δυνατό.

Ωστόσο καθώς σε κάποιες εφαρμογές μπορεί αυτή η προεπιλεγμένη συμπεριφορά να μην είναι η κατάλληλη, δόθηκε η δυνατότητα πλήρους έλεγχου στον προγραμματιστή ώστε να μπορεί να ελέγξει αν μία ημερομηνία ανήκει σε κενό ή επικάλυψη και να ορίσει την δική του επιθυμητή συμπεριφορά.

4.7.3 Άλμα δευτερολέπτου (Leap Second)

Το άλμα δευτερολέπτου στην Συγχρονισμένη Παγκόσμια Ώρα αποτελεί ένα ιδιαίτερο ζήτημα στην υλοποίησή του. Κατά καιρούς έχουν προταθεί διάφορες λύσεις για την υλοποίησή του ή καλύτερα

την πρόβλεψή του στα διάφορα υπολογιστικά συστήματα. Δυστυχώς καμία από αυτές δεν αποτελεί ιδανική λύση καθώς πάντα πρέπει να θυσιαστεί κάτι, για παράδειγμα η ακρίβεια ή η συμβατότητα με άλλα συστήματα.

Η προσέγγιση που ακολουθήθηκε στην υλοποίηση της βιβλιοθήκης είναι παρόμοια με αυτή των προδιαγραφών JSR-000310. Ορίστηκε μία ξεχωριστή κλίμακα ώρας στην οποία τις ημέρες που δεν έχουμε άλμα δευτερολέπτου η ώρα αντιστοιχεί στην Συγχρονισμένη Παγκόσμια Ώρα.

Στις ημέρες που εφαρμόζεται το άλμα δευτερολέπτου, ο ορισμός της κλίμακας θέτει το μεσημέρι να αντιστοιχεί στην ίδια ώρα με την Συγχρονισμένη Παγκόσμια Ώρα, ενώ οι υπόλοιπες χρονικές στιγμές αντιστοιχούν με όσο μεγαλύτερη ακρίβεια γίνεται, μοιράζοντας ουσιαστικά το επιπλέον ή το λιγότερο δευτερόλεπτο σε κάποια από τα δευτερόλεπτα της ημέρας αυτής.

Με αυτόν τον τρόπο επιτυγχάνεται η ημέρα να έχει 86400 δευτερόλεπτα και να μην διαταράσσονται οι υπολογισμοί καθώς και οι συμβατότητα με άλλα συστήματα από το άλμα του δευτερολέπτου. Όμως αυτό οδηγεί σε μη ακριβή μέτρηση των χρονικών στιγμών.

4.7.4 Χρονολογίες

Παρότι το Γρηγοριανό ημερολόγιο έχει υιοθετηθεί από την πλειοψηφία των χωρών του κόσμου, σε αρκετές περιπτώσεις χρησιμοποιούνται διαφορετικά ημερολόγια που είτε το υλοποιούν μερικώς είτε καθόλου. Αυτό δημιουργεί ένα ακόμα ζήτημα για την υλοποίηση μίας βιβλιοθήκης ημερομηνίας και ώρας.

Η λύση που επιλέχθηκε στο πρόβλημα αυτό ήταν να χρησιμοποιηθεί ως βάση το Γρηγοριανό ημερολόγιο και να δώσει την δυνατότητα επέκτασης της βιβλιοθήκης για άλλα ημερολόγια. Αυτό το επιτυγχάνει με την έννοια της χρονολογίας όπου ορίζει ένα σύνολο εννοιών και λειτουργιών τις οποίες θα πρέπει ο προγραμματιστής να υλοποιήσει όπου έχουν νόημα ώστε να δημιουργήσει το επιθυμητό ημερολόγιο.

Κατά κανόνα κομμάτια της λειτουργικότητας του Γρηγοριανού ημερολογίου δεν είναι συμβατά με άλλα ημερολόγια, έτσι για να αποφεύγονται σφάλματα, προτείνεται οι πράξεις με ημερομηνίες και ώρες να γίνονται στο Γρηγοριανό ημερολόγιο και κατά την απεικόνιση να μετατρέπονται οι ημερομηνίες στο επιθυμητό ημερολόγιο.

Κεφάλαιο 5

Συμπεράσματα

5.1 Συνεισφορά

Με την εργασία αυτή υλοποιήθηκε μία βιβλιοθήκη ημερομηνίας και ώρας για την γλώσσα προγραμματισμού Rust. Η βιβλιοθήκη αυτή προσφέρει στον προγραμματιστή της Rust ένα εκτενές μοντέλο των εννοιών του χρόνου και της λειτουργικότητάς τους.

Κάνοντας χρήση της βιβλιοθήκης μπορεί κάποιος να εκφράσει, ακολουθώντας το πρότυπο ISO 8601, ημερομηνίες, ώρες, τον συνδυασμό αυτών και με ή χωρίς εξάρτηση από τις ζώνες ώρας. Επίσης μπορεί να εκφράσει περιόδους και διαστήματα, καθώς και να ορίσει την τρέχουσα ώρα για το πρόγραμμά του.

Η εκφραστικότητα αυτή επιτρέπει στον προγραμματιστή να χρησιμοποιεί κάθε φορά τις δομές που είναι κατάλληλες για την εφαρμογή του. Με αυτόν τον τρόπο δεν χρειάζεται να χειρίζεται δομές με περιττά στοιχεία και έτσι μπορεί να αποφύγει ζητήματα ασφάλειας και απόδοσης κατά την εκτέλεση του προγράμματός του καθώς και να βελτιώσει την αναγνωσιμότητα του κώδικα του.

Πέρα όμως από τις παραπάνω δομές η βιβλιοθήκη δίνει την δυνατότητα για πράξεις με τις ημερομηνίες και τις ώρες. Μπορεί κάποιος να προσθέσει ή να αφαιρέσει ποσότητες χρόνου, να μετακινηθεί μπρος και πίσω στο χρόνο με ασφάλεια, να ορίσει τις μετακινήσεις αυτές και ακόμα να ρυθμίσει τα χρονικά μεγέθη με συγκεκριμένες συναρτήσεις.

Με αυτόν τον τρόπο δημιουργείται ένα σύνολο μεθόδων και συναρτήσεων χειρισμού του χρόνου που καλύπτουν τις βασικές ανάγκες ενός χρήστη της βιβλιοθήκης και η λειτουργία τους είναι σαφώς ορισμένη και σε αναγνώσιμη μορφή.

Τέλος δίνεται η δυνατότητα, μέσω του γνωρίσματος της χρονολογίας, της επέκτασης της βιβλιοθήκης με διάφορες άλλες χρονολογίες που βασίζονται σε διαφορετικά ημερολόγια από το Γρηγοριανό στο οποίο αναφέρεται το πρότυπο ISO 8601.

Καθώς παγκοσμίως η μέτρηση του χρόνου δεν γίνεται με τις ίδιες μεθόδους και τα ίδια εργαλεία, η επεκτασιμότητα της βιβλιοθήκης βοηθά ώστε να μπορεί ο προγραμματιστής να υλοποιήσει τις προδιαγραφές που περιγράφουν τις ανάγκες του, εκμεταλλευόμενος την ήδη υπάρχουσα λειτουργικότητα της βιβλιοθήκης.

5.2 Μελλοντική Εργασία

Μελλοντικά υπάρχει η πρόθεση υλοποίησης κάποιων ακόμα εννοιών και λειτουργιών που τις συνοδεύουν, με σκοπό να αυξηθεί η λειτουργικότητα και η χρηστικότητα της βιβλιοθήκης αυτής.

Πιο συγκεκριμένα, η επέκταση της βιβλιοθήκης με την υλοποίηση χρονολογιών που χρησιμοποιούνται από σημαντικό μέρος του παγκόσμιου πληθυσμού, όπως για παράδειγμα το ισλαμικό ημερολόγιο ή το ιαπωνικό ημερολόγιο.

Σχεδιάζεται επίσης η ενσωμάτωση ποσοτήτων όπως ημέρες, εβδομάδες, μήνες και έτη, καθώς και εννοιών όπως της ημέρας του μήνα, της ημέρας του έτους, του τετραμήνου του έτους.

Τέλος μία ακόμα λειτουργία που κρίνεται απαραίτητη να αποτελέσει μέρος της βιβλιοθήκης είναι η υποστήριξη κάποιων βασικών μοτίβων/μορφοποιήσεων, καθώς και η δυνατότητα δημιουργίας νέων, για την ανάγνωση και την εκτύπωση/εγγραφή των διάφορων εννοιών.

Βιβλιογραφία

[1] Oracle Corporation and/or its affiliates, "JSR 310: Date and Time API", `https://jcp.org/en/jsr/detail?id=310`. Accessed: 2015-10-12.

[2] S. Allen, "The Future of Leap Seconds in UTC", `http://www.ucolick.org/~sla/leapsecs/`. Accessed: 2015-10-16.

[3] Bonnie Blackburn, *The Oxford companion to the year*, Oxford University Press, Oxford New York, 1999.

[4] L. Essen, "Time Scales", *Metrologia*, vol. 4, pp. 161–165, October 1968.

[5] D. Finkleman, S. Allen, J. Seago, R. Seaman and P. K. Seidelmann, "The Future of Time: UTC and the Leap Second", *ArXiv e-prints*, June 2011.

[6] IANA, "Time Zone Database", `https://doc.rust-lang.org/stable/book/`. Accessed: 2015-10-11.

[7] ISO, "Data elements and interchange formats – Information interchange – Representation of dates and times", ISO 8601:2004, International Organization for Standardization, Geneva, Switzerland, 2004.

[8] S. Leschiutta, "The definition of the 'atomic' second", *Metrologia*, vol. 42, p. 10, June 2005.

[9] A. Philip, *The Calendar: Its History, Structure and Improvement*, Cambridge University Press, 2012.

[10] "The Rust Programming Language", `http://www.iana.org/time-zones`. Accessed: 2015-10-14.

[11] P. Schlyter, "Timescales", `http://www.stjarnhimlen.se/comp/time.html`. Accessed: 2015-10-13.

[12] IEEE Computer Society, "IEEE Standard for Floating-Point Arithmetic", Technical report, IEEE, 2008.

[13] G. J. Whitrow, *Time in history : views of time from prehistory to the present day*, Oxford University Press, Oxford New York, 1989.

[14] Wikipedia, "Time zone — Wikipedia, The Free Encyclopedia", 2015. [Online; accessed 20-October-2015].