Hinweise zur Bearbeitung und Abgabe

Die Lösung der Hausaufgabe muss **eigenständig** erstellt werden. Abgaben, die identisch oder auffällig ähnlich zu anderen Abgaben sind, werden als **Plagiat** gewertet! **Plagiate sind Täuschungsversuche und führen zur Bewertung „nicht bestanden" für die gesamte Modulprüfung.**

- Bitte nutzen Sie MARS zum Simulieren Ihrer Lösung. Stellen Sie sicher, dass Ihre Abgabe in MARS ausgeführt werden kann.
- Sie erhalten für jede Aufgabe eine separate Datei, die aus einem Vorgabe- und Lösungsabschnitt besteht. Ergänzen Sie bitte Ihren Namen und Ihre Matrikelnummer an der vorgegebenen Stelle. Bearbeiten Sie zur Lösung der Aufgabe nur den Lösungsteil unterhalb der Markierung:
  ```
  #+ Loesungsabschnitt
  #+ ------------------
  ```
- Ihre Lösung muss auch mit anderen Eingabewerten als den vorgegebenen funktionieren. Um Ihren Code mit anderen Eingaben zu testen, können Sie die Beispieldaten im Lösungsteil verändern.
- Bitte nehmen Sie keine Modifikationen am Vorgabeabschnitt vor und lassen Sie die vorgegebenen Markierungen (Zeilen beginnend mit `#+`) unverändert.
- Eine korrekte Lösung muss die bekannten **Registerkonventionen** einhalten. Häufig können trotz nicht eingehaltener Registerkonventionen korrekte Ergebnisse geliefert werden. In diesem Fall werden trotzdem Punkte abgezogen.
- Falls Sie in Ihrer Lösung zusätzliche Speicherbereiche für Daten nutzen möchten, verwenden Sie dafür bitte ausschließlich den **Stack** und keine statischen Daten in den Datensektionen (`.data`). Die Nutzung des Stacks ist gegebenenfalls notwendig, um die Registerkonventionen einzuhalten.
- Die zu implementierenden Funktionen müssen als Eingaben die Werte in den **Argument-Registern** (`$a0`–`$a3`) nutzen. Daten in den Datensektionen der Assemblerdatei dürfen nicht direkt mit deren Labels referenziert werden.
- Bitte gestalten Sie Ihren Assemblercode nachvollziehbar und verwenden Sie detaillierte Kommentare, um die Funktionsweise Ihres Assemblercodes darzulegen.
- Die Abgabe erfolgt über ISIS. Laden Sie die zwei Abgabedateien separat hoch.

Aufgabe 1: Morse-Dekoder (10 Punkte)

Hintergrund: Morsecode-Nachrichten bestehen aus Punkten, Strichen und Leerzeichen. Folgen von Punkten und Strichen kodieren einzelne Zeichen. Leerzeichen trennen aufeinanderfolgende Zeichen voneinander. Um ein einzelnes Zeichen zu dekodieren, wird der in Abbildung 1 gezeigte Binärbaum genutzt. Ausgehend vom Startknoten „(0) _" wird bei einem Punkt der linke Kindknoten ausgewählt, bei einem Strich der rechte Kindknoten. „`.-`" kodiert beispielsweise das Zeichen „A", „`--..`" das Zeichen „Z".

Aufgabe: Implementieren Sie die Funktion `morse`, welche eine Morsecode-Nachricht `morse_in` in eine Zeichenfolge `text_out` (Buchstaben, Zahlen und Sonderzeichen) umwandelt. Die C-Signatur der zu implementierenden Funktion ist:

`int`	`morse(`	`char* decoder_heap,`	`char *morse_in,`	`char *text_out);`
`$v0`		`$a0`	`$a1`	`$a2`

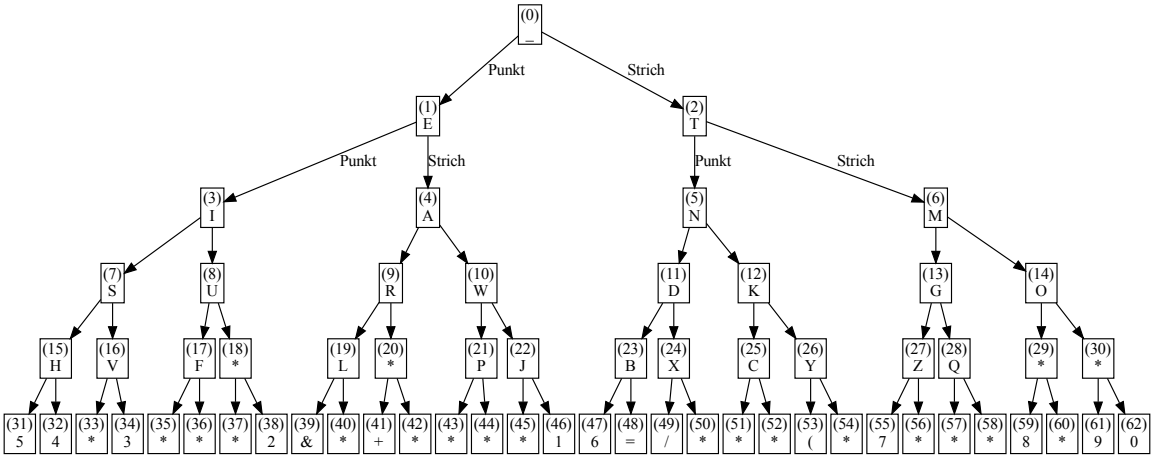

Abbildung 1: Binärbaum zum Dekodieren von Morsecode

Array-Index	0	1	3	7	15	0	1	4	0	1	4	9	19	0	1	4	9	19	0	2	6	14
Eingabezeichen	.	.	.	.	␣	.	-	␣	.	-	.	.	␣	.	-	.	.	␣	-	-	-	'\0'
Ausgabezeichen					H			A					L					L				O

Abbildung 2: Beispielhaftes Vorgehen des Morse-Dekoders (␣= Leerzeichen)

Bei `morse_in` handelt es sich um einen Pointer zum ersten Zeichen der Morsecode-Nachricht. Die Morsecode-Nachricht enthält die ASCII-Zeichen `'.'` (Punkt), `'-'` (Strich) und `' '` (Leerzeichen) und endet mit einem Nullterminator (`'\0'`). Der dekodierte Text soll als Zeichenkette beginnend an der Adresse `text_out` abgelegt werden und mit einem Nullterminator abgeschlossen werden. Die Funktion `morse` soll außerdem die Zahl der dekodierten Zeichen zurückgeben.

Decoder-Baum als Heap: Das Array `decoder_heap`, welches als Funktionsargument übergeben wird, enthält den Decoder-Baum aus Abbildung 1 als *Heap*[1]. In diesem Array sind alle kodierbaren Zeichen aus dem Decoder-Baum in der Reihenfolge *links nach rechts* und *oben nach unten* enthalten. Die Array-Indizes, die sich daraus ergeben, sind in Abbildung 1 in Klammern angegeben. Diese spezifische Reihenfolge ermöglicht es, den Baum durch einfache Rechenoperationen auf einem Array-Index k zu durchlaufen:

- Um den Wurzelknoten auszuwählen, setze $k := 0$.
- Um zum linken Kindknoten des aktuellen Knotens auszuwählen, setze $k := 2 \cdot k + 1$.
- Um zum rechten Kindknoten des aktuellen Knotens auszuwählen, setze $k := 2 \cdot k + 2$.

Vorgehen: In einer Variable soll die aktuelle Position im Decoder-Heap als Array-Index gespeichert werden. Zu Beginn soll dieser Array-Index auf 0 gesetzt werden. Dann soll die Eingabe `morse_in` Zeichen für Zeichen durchlaufen werden:

- Falls ein Punkt gelesen wird, wird der linke Kindknoten ausgewählt.
- Falls ein Strich gelesen wird, wird der rechte Kindknoten ausgewählt.
- Falls ein Leerzeichen oder der Nullterminator gelesen wird, wird das Zeichen des aktuellen Knotens aus dem Decoder-Heap gelesen und an die Ausgabezeichenkette `text_out` angehängt. Außerdem wird eine Zählvariable für den Rückgabewert erhöht und der Array-Index für den Decoder-Heap wird auf den Wurzelknoten zurückgesetzt. Ein Nullterminator beendet die Dekodierung.

Abbildung 2 verdeutlicht das Vorgehen anhand eines Beispiels.

[1]Weitere Hintergrundinformationen: https://de.wikipedia.org/wiki/Binärer_Heap

Test-Eingaben: Testen Sie Ihre Lösung mit unterschiedlichen Eingaben! Bearbeiten Sie dazu die Zeichenkette `test_msg`. Folgende Tabelle enthält Beispieleingaben und die erwarteten Rückgabewerte, welche zum Testen verwendet werden können:

Morsecode-Nachricht (␣= Leerzeichen)	**Dekodierter Text**	**Rückgabewert**
`....␣.-␣.-..␣.-..␣---`	`HALLO`	5
`--␣..␣.--.␣...␣␣..␣...␣-␣␣-␣---␣.-..␣.-..`	`MIPS_IST_TOLL`	13
`.----␣.-.-.␣.----␣-...-␣..---`	`1+1=2`	5

Hinweise:

- Sie dürfen annehmen, dass es sich bei `morse_in` immer um eine gültige Morsecode-Nachricht handelt, welche ausschließlich in Abbildung 1 gezeigte Zeichen enthält.
- Die Multiplikation mit 2 kann durch eine logische Schiebeoperation durchgeführt werden.
- Falls zwei Leerzeichen aufeinanderfolgen, soll beim zweiten Leerzeichen das Zeichen des Wurzelknotens (`'_'`) ausgegeben werden. Dies entspricht der Idee, dass Wörter in Morsecode durch längere Pausen kodiert werden (siehe Beispiel `MIPS_IST_TOLL`).
- Sie dürfen annehmen, dass `text_out` immer groß genug ist, um den gesamten dekodierten Text abspeichern zu können.

Aufgabe 2: Römische Zahlen dekodieren (10 Punkte)

Hintergrund: Römische Zahlen setzen sich aus römischen Buchstaben zusammen, welche bestimmte Zahlenwerte repräsentieren: `I` (1), `V` (5), `X` (10), `L` (50), `C` (100), `D` (500), `M` (1000). Sofern eine Ziffer nicht direkt vor einer Ziffer mit größerem Zahlenwert steht, ist sie Summand mit positivem Vorzeichen (Beispiel: `LXXII` $= 50 + 10 + 10 + 1 + 1 = 72$). Eine Ziffer, die direkt vor einer Ziffer mit größerem Zahlenwert steht, ist als Summand mit negativem Vorzeichen zu verstehen (Beispiel: `XIV` $= 10 - 1 + 5 = 14$).

Aufgabe: Implementieren Sie die Funktion `roman`. Diese Funktion soll die übergebene römische Zahl `numeral` dekodieren und das Umwandlungsergebnis zurückgeben. Die C-Signatur der zu implementierenden Funktion lautet:

`int roman( char *numeral);`
`$v0` `$a0`

Zur Lösung der Aufgabe **muss** die **Hilfsfunktion** `romdigit` genutzt werden. `romdigit` wandelt eine **einzelne römische Ziffer** in den entsprechenden Zahlenwert um. Falls es sich bei der übergebenen Ziffer nicht um eine römische Ziffer handelt, gibt `romdigit` 0 zurück. C-Signatur der Hilfsfunktion:

`int romdigit( char digit);`
`$v0` `$a0`

Vorgehen: Durchlaufen Sie die Zeichenkette `numeral` von vorn nach hinten und dekodieren Sie die einzelnen Zeichen durch Aufruf der Hilfsfunktion `romdigit`. Vergleichen Sie den Zahlenwert jeder Ziffer mit dem Zahlenwert der darauf folgenden Ziffer. Abhängig von diesem Vergleich muss der Zahlenwert zu dem Gesamtergebnis hinzugezählt oder vom Gesamtergebnis abgezogen werden. Der Zahlenwert der letzten Ziffer wird immer zum Gesamtergebnis hinzugezählt.

Test-Eingaben: Testen Sie Ihre Lösung mit unterschiedlichen Eingaben! Bearbeiten Sie dazu die Zeichenkette `test_numeral`. Folgende Tabelle enthält Beispieleingaben und die erwarteten Rückgabewerte:

Römische Zahl	`III`	`L`	`XI`	`XIV`	`CLXXXIV`	`DCCXLVI`	`MMMCMXCIX`
Erwarteter Rückgabewert	3	50	11	14	184	746	3999

Hinweise:

- Sie dürfen annehmen, dass es sich bei jeder übergebenen Zahl `numeral` um eine gültige römische Zahl zwischen `I` (1) und `MMMCMXCIX` (3999) handelt. Es muss also nicht geprüft werden, ob es sich bei der Eingabe um eine gültige römische Zahl handelt. Bei ungültiger Eingabe darf `roman` einen beliebigen Rückgabewert liefern.
- Sie können ausnutzen, dass `romdigit` für den Nullterminator (Wert 0) den Wert 0 zurückgibt.
- In römischen Zahlen folgen nie mehrere negative Summanden aufeinander. Beispiele: 8 wird als `VIII` kodiert, nicht als `IIX`; 89 wird als `LXXXIX` kodiert, nicht als `IXC`.