Rechnerorganisation SS18
Testat P.1

TECHNISCHE UNIVERSITÄT DARMSTADT

Prof. Dr.-Ing. Michael Goesele, M.Sc. Daniel Thürck und Tobias Stensbeck
Version 2 vom 22. Mai 2018

Hinweis: Die Abgabe ihres Codes erfolgt per Moodle; die Abgabefrist ist der 03.06. um 23:30 Uhr. Näheres zur anschließenden Besprechung Ihrer Ergebnisse mit Ihrer Tutorin bzw. Ihrem Tutor wird in Kürze bekannt gegeben. Dieses Praxistestat soll individuell bearbeitet und abgegeben werden; es gelten die üblichen Richtlinien zum Plagiarismus des Fachbereichs 20, siehe `https://www.informatik.tu-darmstadt.de/studium_fb20/im_studium/studienbuero/plagiarismus/index.de.jsp`.

Einführung

Hinweis: Verwenden Sie die **bereitgestellte Vorlage** und bearbeiten Sie die Aufgaben in der **angegebenen Reihenfolge**, da diese aufeinander aufbauen. Lesen Sie sich zunächst die ganze Aufgabenstellung durch, um einen Überblick zu erhalten.
Nehmen Sie sich **ausreichend Zeit** zur Bearbeitung und kommentieren Sie Ihren Code zwecks späterer Nachvollziehbarkeit.

Ziel dieses Testates ist, einen interaktiven Taschenrechner zu programmieren. Schreiben Sie dazu ein MIPS-Programm, das einfache arithmetische Ausdrücke wie z. B. "`42+1*2*3`" oder "`4*(5+6)`" auswertet. Das Programm soll Integer-Arithmetik verwenden (d. h. das Ergebnis von `3/2` soll `1` sein).

Ein arithmetischer Ausdruck besteht aus positiven Zahlliteralen (z. B. "`42`"), den 4 üblichen Operatoren für die Grundrechenarten ("+", "-", "*", "/") und Klammern ("(", ")").

Es sind nur mathematisch gültige Ausdrücke zulässig. Vorzeichen ("`-32`") und unäre Minuszeichen ("`1+(-2)`") müssen **nicht** behandelt werden und diese Ausdrücke sind als fehlerhaft zurückzuweisen.

Das Programm soll interaktiv Eingaben von der Konsole einlesen. Sobald der Benutzer eine Zeile eingegeben hat und die Entertaste drückt, soll diese ausgewertet werden. Anschließend wird für gültige Ausdrücke das Ergebnis des Ausdrucks ausgegeben. Wurde eine ungültige Eingabe gemacht, dann soll das Programm eine Fehlermeldung ausgeben.

```
> 1+2+3
6
> 42+1*2*3
48
> 4*(5+6)
44
> 4)+5
Error
```

Beispiel für die Ausgabe des Programms. Zeilen, die mit ">" beginnen, stellen Benutzereingaben dar.

Es steht Ihnen eine Vorlage zur Verfügung, in der das Einlesen der Benutzereingaben bereits implementiert ist. Die Vorlage liest immer eine Zeile ein und ruft dann die Funktion `handle_one_line` auf, in der Sie Ihre eigene Logik implementieren können. Außerdem verwaltet die Vorlage einen Cursor, der auf ein bestimmtes Zeichen in der Benutzereingabe zeigt. Damit können die einzelnen Funktionen effektiv kommunizieren, welcher Teil der Eingabe bereits verarbeitet wurde. Bevor `handle_one_line` aufgerufen wird, wird der Cursor von der Vorlage zurückgesetzt, sodass er auf das erste Zeichen der Benutzereingabe zeigt. Sie können den Cursor dann Zeichen für

Zeichen von links nach rechts durch die Benutzereingabe wandern lassen. Das Ende der aktuellen Zeile wird durch einen Line Feed (ASCII 10) markiert. Die folgenden Funktionen bzw. Prozeduren sind in der Vorlage implementiert.

- **get_current_char** gibt das aktuelle Zeichen (das Zeichen auf das der Cursor gerade zeigt) in `$v0` zurück. Diese Funktion verändert den Cursor nicht. Wenn Sie die Funktion mehrmals hintereinander aufrufen würden, würden Sie immer das gleiche Ergebnis erhalten.
- **advance_cursor** schiebt den Cursor ein Zeichen weiter.
- **print_status** Gibt die Werte von `$v0` und $v1 sowie die aktuelle Position des Cursors aus.

Wichtig: Befolgen Sie die MIPS Calling Conventions, die in der Vorlesung vorgestellt worden sind, wenn Sie neue Funktionen hinzufügen. Um Ihre Funktionen zu testen, rufen Sie diese aus `handle_one_line` heraus auf und lassen Sie sich das Ergebnis über die Funktion `print_status` ausgeben. Sie sollten nach jeder Aufgabe ein lauffähiges Programm haben. Testen Sie Ihre Funktionen bevor Sie mit der nächsten Aufgabe fortfahren.

Aufgabe 1.1 Ziffern erkennen

Eine Ziffer ist eines der Zeichen “0” bis “9”.

Schreiben Sie eine Funktion **is_num**, die erkennt ob es sich bei dem aktuellen Zeichen um eine Ziffer handelt oder nicht. Die Funktion soll eine 1 in `$v0` zurückgeben, wenn das aktuelle Zeichen eine Ziffer ist. Andernfalls ist eine 0 zurückzugeben.

Rufen Sie die Funktion `get_current_char`, die in der Vorlage implementiert ist, auf, um das aktuelle Zeichen abzufragen. Weiter unten finden Sie eine ASCII-Tabelle, in der Sie die ASCII-Werte der relevanten Zeichen nachschauen können.

Ist die Benutzereingabe z. B. “1” oder “2” sollte in `$v0` eine 1 stehen. Ist die Benutzereingabe z. B. ein “!” sollte in `$v0` eine 0 stehen. Der Cursor sollte seine Position nicht verändern.

Aufgabe 1.2 Zahlen verarbeiten

Eine Zahl besteht aus einer beliebigen Anzahl an Ziffern.

Schreiben Sie eine Funktion **read_num**, die eine Zahl einliest und ihren numerischen Wert zurückgibt. Die Funktion soll so viele Zeichen wie möglich verarbeiten und das Ergebnis in `$v0` schreiben.

Benutzen Sie die Funktion `is_num` aus der vorherigen Aufgabe. Solange das aktuelle Zeichen eine Ziffer ist, soll die Funktion das Zeichen verarbeiten und dann `advance_cursor` aufrufen, um zum nächsten Zeichen überzugehen. Wenn das aktuelle Zeichen keine Ziffer ist, soll die Funktion abbrechen und den Wert der verarbeiteten Zeichenkette zurückgeben. Beachten Sie, dass die ASCII-Werte der Ziffern nicht mit ihrem numerischen Wert übereinstimmen.

Ist zum Beispiel die Benutzereingabe “123+567” und der Cursor zeigt auf die “1”, dann soll die Funktion den Wert 123 zurückgeben und der Cursor nach dem Funktionsaufruf auf das “+” zeigen.

Aufgabe 1.3 Faktoren und Fehlererkennung

Ein Faktor ist eine Zahl oder ein geklammerter Ausdruck. Da wir geklammerte Ausdrücke erst in Aufgabe 1.6 behandeln werden, gehen Sie für diese Aufgabe davon aus, dass ein Faktor nur eine Zahl sein kann.

Schreiben Sie eine Funktion **parse_factor**, die versucht einen Faktor einzulesen. Diese Funktion soll den numerischen Wert des Faktors in `$v0` und einen Fehlerstatus in $v1 zurückgeben. Als Fehlerstatus soll 0 zurückgegeben werden, wenn ein Faktor eingelesen werden konnte, und 1 zurückgegeben werden, wenn es einen Fehler gab.

Analysieren Sie in Ihrer Funktion das erste Zeichen. Ist es eine Ziffer, rufen Sie die Funktion `read_num` aus der vorherigen Aufgabe auf. Andernfalls geben Sie als Wert 0 und als Fehlerstatus 1 zurück.

Ist die Benutzereingabe “123+567” sollte wieder 123 und Fehlerstatus 0 zurückgegeben werden. Die übrigen Zeichen “+567” werden von dieser Funktion ignoriert. Ist die Benutzereingabe z. B. “?!?” sollte Fehlerstatus 1 zurückgegeben werden.

Aufgabe 1.4 Erste Rechenoperationen

Als Term bezeichnen wir in diesem Testat einen oder mehrere Faktoren, die über Multiplikation oder Division verknüpft sind. Ein einzelner Faktor (“123”) ist also ebenfalls ein Term. Weitere Beispiele für Terme sind “123*567” oder auch “1*2/3*4”.

Schreiben Sie eine Funktion **parse_term**, die einen Term auswertet. Die Funktion soll die gleichen Rückgabewerte haben wie `parse_factor`, d. h. `$v0` soll den numerischen Wert des Termes enthalten und `$v1` soll einen Fehlerstatus (0 - kein Fehler, 1 - Fehler) enthalten.

Die Funktion soll also eine Reihe von Multiplikationen und Divisionen verarbeiten. Benutzen Sie zunächst die Funktion `parse_factor`, um einen Faktor einzulesen. Folgt anschließend eines der Zeichen “*” oder “/”, lesen Sie es ein und rufen Sie `parse_factor` erneut auf, um einen weiteren Faktor einzulesen. Führen Sie dann die entsprechende Rechenoperation auf den beiden Faktoren aus. Folgt ein weiterer “*” oder “/”, wiederholen Sie den Prozess und führen Sie die Rechenoperation auf dem Ergebnis des vorherigen Schrittes und dem Wert des neuen Faktors aus. Kommt nach einem Faktor ein anderes Zeichen, geben Sie das aktuelle Ergebnis und Fehlerstatus 0 zurück. Falls `parse_factor` einen Fehler zurückliefert, soll die Funktion ebenfalls Wert 0 und Fehlerstatus 1 zurückgeben. Wie bereits erwähnt soll Integer-Arithmetik verwendet werden.

Ihr Programm sollte nun Benutzereingaben wie “1*2*3*4” oder “5*6/3” auswerten können. Unzulässige Eingaben wie z. B. “1**2” oder “1*!” sollten zu Fehlerstatus 1 führen. Bei der Eingabe “1*2+3*4” sollte Ergebnis 2 und Fehlerstatus 0 zurückgegeben werden, da die ersten drei Zeichen eine gültige Multiplikation sind. Auch die Eingabe “1*2!!!” sollte Ergebnis 2 und Fehlerstatus 0 erzeugen, da die Eingabe wieder mit einer gültigen Multiplikation beginnt - der Cursor zeigt danach auf das erste “!”. Der Rest der Eingabe nach dem letzten erkannten Term kann ignoriert werden.

Aufgabe 1.5 Addition und Subtraktion

Ein Ausdruck ist ein Term oder besteht aus mehreren Termen, die über Addition oder Subtraktion verknüpft sind.

Schreiben Sie eine weitere Funktion **parse_expression**, die einen Ausdruck verarbeitet. Auch diese Funktion soll sich an die gleichen Rückgabekonventionen wie `parse_factor` und `parse_term` halten.

Benutzen Sie `parse_term`, um Terme einzulesen und führen Sie dann analog zur Implementierung von `parse_term` eine Reihe von Additionen und Subtraktionen auf den Werten aus. Durch diese Staffelung der Funktionen werden automatisch die Operatorpräzedenzen eingehalten, da alle Multiplikationen und Divisionen in `parse_term` ausgeführt werden, bevor das Ergebnis mit den Werten von anderen Termen addiert bzw. subtrahiert wird.

Sie sollten nun zusammengesetze Ausdrücke verarbeiten können: “1+2*3”, “4+6/6+5”. Analog zu `parse_term` dürfen ungültige Zeichen nach der letzten gültigen Expression ignoriert werden.

Aufgabe 1.6 Klammerung

Wir sind nun in der Lage, geklammerte Ausdrücke zu verarbeiten. Ein geklammerter Ausdruck besteht aus einer öffnenden Klammer, gefolgt von einem Ausdruck und abschließend einer schließenden Klammer.

Erweitern Sie nun die Funktion **parse_factor** um einen weiteren Fall. Ist das aktuelle Zeichen eine öffnende Klammer, dann soll diese Klammer eingelesen und anschließend `parse_expression` aufgerufen werden. Abschließend muss eine schließende Klammer eingelesen werden. Falls keine schließende Klammer folgt, ist das ein Fehler. Andernfalls geben Sie das Ergebnis und den Fehlerstatus von `parse_expression` zurück. Ist das aktuelle Zeichen weder eine Ziffer noch eine öffnende Klammer, dann soll weiterhin Wert 0 und Fehlerstatus 1 zurückgegeben werden.

Testen Sie Ihre Implementierungen mit Eingaben wie z. B. “(1+2)” = 3, “2*(1+2)” = 6, “1*(1+2*3)*3” = 21, “(1*((3+3)*(3+4)))” = 42. Testen Sie auch fehlerhafte Eingaben wie z. B. “()” oder “(1”.

Aufgabe 1.7 Abschluss

Der Taschenrechner ist nun vollständig. Schreiben Sie nun **`handle_one_line`** um, sodass der Fehlerstatus, der von `parse_expression` zurückgegeben wird, ausgewertet wird. Ist der Fehlerstatus 0, dann geben Sie den erhaltenen Wert aus. Ist der Fehlerstatus 1, dann geben Sie die Nachricht “Error” aus.

Hinweise zur Implementierung

ASCII-Tabelle[2]

Dec	Hex	Char	Name	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0x00	NUL	Null	32	0x20	SPACE	64	0x40	@	96	0x60	‘
1	0x01	SOH	Start of Heading	33	0x21	!	65	0x41	A	97	0x61	a
2	0x02	STX	Start of Text	34	0x22	"	66	0x42	B	98	0x62	b
3	0x03	ETX	End of Text	35	0x23	#	67	0x43	C	99	0x63	c
4	0x04	EOT	End of Transmission	36	0x24	$	68	0x44	D	100	0x64	d
5	0x05	ENQ	Enquiry	37	0x25	%	69	0x45	E	101	0x65	e
6	0x06	ACK	Acknowledgement	38	0x26	&	70	0x46	F	102	0x66	f
7	0x07	BEL	Bell	39	0x27	’	71	0x47	G	103	0x67	g
8	0x08	BS	Backspace	40	0x28	(	72	0x48	H	104	0x68	h
9	0x09	HT	Horizontal Tab	41	0x29	)	73	0x49	I	105	0x69	i
10	0x0a	LF	Line Feed	42	0x2a	∗	74	0x4a	J	106	0x6a	j
11	0x0b	VT	Vertical Tab	43	0x2b	+	75	0x4b	K	107	0x6b	k
12	0x0c	FF	Form Feed	44	0x2c	,	76	0x4c	L	108	0x6c	l
13	0x0d	CR	Carriage Return	45	0x2d	−	77	0x4d	M	109	0x6d	m
14	0x0e	SO	Shift Out	46	0x2e	.	78	0x4e	N	110	0x6e	n
15	0x0f	SI	Shift In	47	0x2f	/	79	0x4f	O	111	0x6f	o
16	0x10	DLE	Data Link Escape	48	0x30	0	80	0x50	P	112	0x70	p
17	0x11	DC1	Device Control 1	49	0x31	1	81	0x51	Q	113	0x71	q
18	0x12	DC2	Device Control 2	50	0x32	2	82	0x52	R	114	0x72	r
19	0x13	DC3	Device Control 3	51	0x33	3	83	0x53	S	115	0x73	s
20	0x14	DC4	Device Control 4	52	0x34	4	84	0x54	T	116	0x74	t
21	0x15	NAK	Negative Ack.	53	0x35	5	85	0x55	U	117	0x75	u
22	0x16	SYN	Synchronous Idle	54	0x36	6	86	0x56	V	118	0x76	v
23	0x17	ETB	End of Trans. Block	55	0x37	7	87	0x57	W	119	0x77	w
24	0x18	CAN	Cancel	56	0x38	8	88	0x58	X	120	0x78	x
25	0x19	EM	End of Medium	57	0x39	9	89	0x59	Y	121	0x79	y
26	0x1a	SUB	Substitute	58	0x3a	:	90	0x5a	Z	122	0x7a	z
27	0x1b	ESC	Escape	59	0x3b	;	91	0x5b	[	123	0x7b	{
28	0x1c	FS	File Separator	60	0x3c	<	92	0x5c	\	124	0x7c	|
29	0x1d	GS	Group Separator	61	0x3d	=	93	0x5d	]	125	0x7d	}
30	0x1e	RS	Record Separator	62	0x3e	>	94	0x5e	^	126	0x7e	∼
31	0x1f	US	Unit Separator	63	0x3f	?	95	0x5f	_	127	0x7f	DEL

System calls

Um auf Funktionen zuzugreifen, die vom Betriebssystem oder in unserem Fall vom MIPS Simulator bereit gestellt werden, werden System Calls verwendet. Dazu wird in Register `$v0` ein Code abgelegt, der die aufzurufende Funktion identifiziert. Außerdem werden eventuelle Argumente in die entsprechenden Register geladen. Anschließend wird die `syscall` Instruktion aufgerufen, die die Ausführung der Funktion anstößt.

[2] Quelle: Wikipedia ASCII https://en.wikipedia.org/wiki/ASCII

Für das Testat sind vor allem die System Calls 1 (print integer) , 11 (print character) und 4 (print string) von Relevanz.

Weitere Informationen sowie eine Liste mit allen verfügbaren System Calls finden Sie in der eingebauten Hilfe vom Mars Simulator (Tab “Syscalls”).