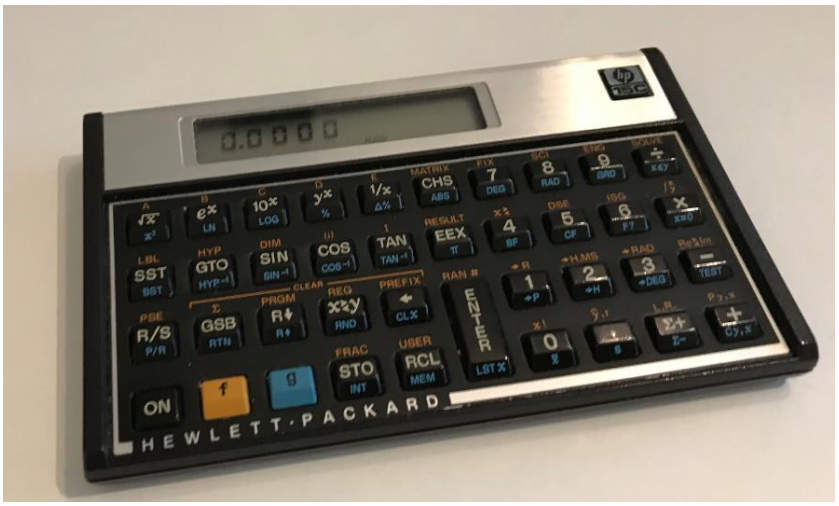
SNP: Integer Rechner



1 Übersicht

In diesem Praktikum erweitern Sie einen Programm Rahmen zu einem funktionierenden Integer Rechner. Der Rechner nimmt **Infix-Ausdrücke** an, konvertiert sie in **UPN Ausdrücke** und berechnet das **Unsigned Integer Resultat**.

Z.B. aus `4 - (1 - 2) * -3` wird `4 1 2 SUB 3 CHS MUL SUB` mit Resultat `1`.

Oder aus `0xFF & ~(0xFF << 4) | (0x55 << 4)`
wird `0xFF 0xFF 4 LSHIFT INV AND 0x55 4 LSHIFT OR` mit Resultat `0x0000055F`.

Ihre Aufgabe ist es, den Stack für die Auswertung der generierten UPN Ausdrücke plus einige der Operatoren zu implementieren.

1.1 Programm Aufruf

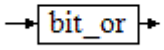
```
echo '0xFF & ~(0xFF << 4) | (0x55 << 4)' | bin/integer-rechner 1
...
--- RESULT ---
hex=0x0000055f
```

Der Ausdruck wird auf **`stdin`** erwartet. Mit einem Argument ungleich 0 wird zusätzlicher Output auf **`stderr`** ausgegeben.

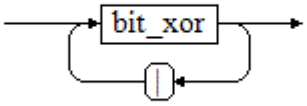
1.2 Infix Syntax

Die Syntax für die Infix Ausdrücke ist ein Subset der C Expression Syntax (gleiche Priorisierung und gleiche Assoziativität). Zusätzlich können Binäre Werte eingegeben werden und bei den Werten zur besseren Lesbarkeit Underscores eingestreut werden. Gerechnet wird intern nur mit Unsigned Werten, was bei Division und Modulo von «negativen» Zahlen zu unerwarteten Resultaten führen kann.

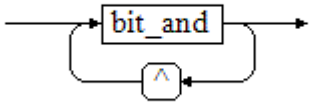
expression



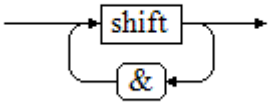
bit_or



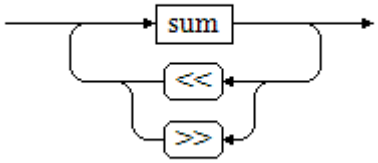
bit_xor



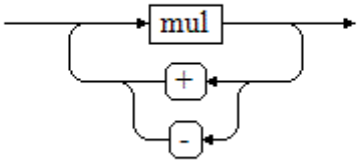
bit_and



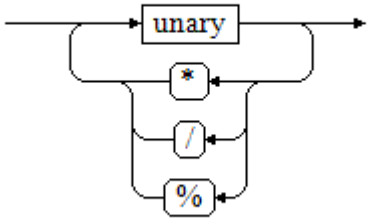
shift



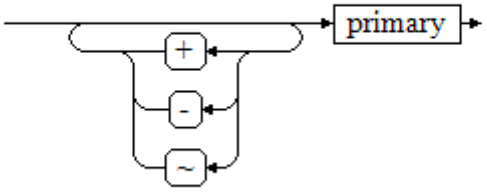
sum



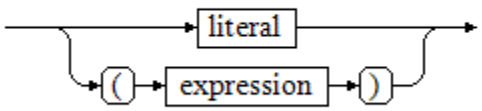
mul



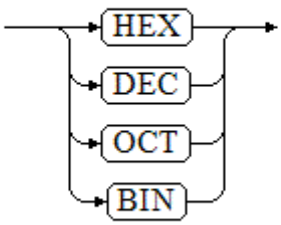
unary



primary



literal



1.3 Übersetzung von Infix zu Postfix

Die Zahlen Werte und Operatoren werden vom Infix Parser in der UPN Reihenfolge an den Evaluator weitergegeben. Dieser legt die ankommenden Zahlen in einem Stack Container ab.

Wenn ein Operator ankommt, wird je nach Operation keiner, einer oder zwei Werte vom Stack gelesen, die Werte gemäss Operator ausgewertet und das Resultat gegebenenfalls wieder auf dem Stack abgelegt. Dies wird repetitiv durchgeführt, bis alle Werte und Operatoren abgearbeitet sind und der Stack nur noch einen Wert beinhaltet, nämlich das Resultat der Auswertung.

Die Übersetzung der Infix Operatoren in Postfix Operatoren ist wie folgt:

Infix Operation	Postfix Operation	Beschreibung
`|`	**`OP_BIT_OR`**	Bit Or
`^`	**`OP_BIT_XOR`**	Bit Xor
`&`	**`OP_BIT_AND`**	Bit And
`<<`	**`OP_BIT_LSHIFT`**	Links Bit Shift
`>>`	**`OP_BIT_RSHIFT`**	Rechts Bit Shift
`+`	**`OP_ADD`**	Summe
`-`	**`OP_SUB`**	Differenz
`*`	**`OP_MUL`**	Unsigned Integer Multiplikation
`/`	**`OP_DIV`**	Unsigned Integer Division
`%`	**`OP_MOD`**	Unsigned Modulo
`+`	n/a	Vorzeichen, No-Operation
`-`	**`OP_CHS`**	Vorzeichen, Change-Sign
`~`	**`OP_INV`**	Einer-Komplement
`( )`	n/a	Gruppierung, No-Operation, durch UPN gegeben
n/a	**`OP_NOP`**	No-Operation
n/a	**`OP_PRINT_HEX`**	Hex Ausgabe der obersten Wertes des Stacks
n/a	**`OP_PRINT_DEC`**	Dezimal Ausgabe der obersten Wertes des Stacks
n/a	**`OP_PRINT_OCT`**	Oktal Ausgabe der obersten Wertes des Stacks
n/a	**`OP_PRINT_BIN`**	Binär Ausgabe der obersten Wertes des Stacks

2 Lernziele

In diesem Praktikum üben Sie mit **Pointers auf Strukturen** zu arbeiten, **Heap Memory** allozieren und freigeben, **Pointer Arithmetik**.

- Sie können anhand einer Beschreibung im Code die fehlenden Funktionen implementieren.

Die Bewertung dieses Praktikums ist am Ende angegeben.

Erweitern Sie die vorgegebenen Code Gerüste, welche im **`git`** Repository **`snp-lab-code`** verfügbar sind.

3 Aufgabe 1: Papierübung Infix zu Postfix Übersetzung

In der folgenden Tabelle sind Infix Ausdrücke mit den entsprechenden Postfix Ausdrücken angegeben. Berechnen Sie von Hand das Resultat für Infix und Postfix (und prüfen Sie auf Gleichheit 😉). Füllen Sie die Stack Werte für jeden Schritt der Berechnung bei Postfix wie in der ersten Zeile gegeben

Infix	Postfix (UPN)	Resultat

`10 - 2  * -3`	`10 : 10`
	`2 : 10, 2`
	`3 : 10, 2, 3`
	`CHS : 10, 2, -3`
	`* : 10, -6`
	`- : 16`
		`16`

`1 + 2 * 3`	`1 :`
	`2 :`
	`3 :`
	`* :`
	`+ :`	

`(1 + 2) * 3`	`1 :`
	`2 :`
	`+ :`
	`3 :`
	`* :`	

`(8 | 7 ^ 5) & 4 << 3 + 2 * -1`	`8 :`
	`7 :`
	`5 :`
	`^ :`
	`| :`
	`4 :`
	`3 :`
	`2 :`
	`1 :`
	`CHS :`
	`* :`
	`+ :`
	`<< :`
	`& :`	

4 Aufgabe 2: Stack

Das zu ergänzende Programm besteht aus den unten aufgeführten Files.

`Makefile`	→ gegeben, d.h. nichts anzupassen (Inkrementelles Builden)
`tests/tests.c`	→ gegeben, d.h. nichts anzupassen (Unit Tests)
`src/main.c`	→ gegeben, d.h. nichts anzupassen (Allozierung, Ausführung, Freigabe)
`src/calc.h`	→ gegeben, d.h. nichts anzupassen (API Infix-Parser)
`src/calc.c`	→ gegeben, d.h. nichts anzupassen (Infix-Parser Implementierung)
`src/error.h`	→ gegeben, d.h. nichts anzupassen (API Error Handling)
`src/erorr.c`	→ gegeben, d.h. nichts anzupassen (Error Handling Implementierung)
`src/eval.h`	→ gegeben, d.h. nichts anzupassen (API UPN Ausführung)
`src/eval.c`	→ **anzupassen**: umsetzen gemäss Angabe 3 (Implementierung)
`src/scan.h`	→ gegeben, d.h. nichts anzupassen (API Infix Tokenizer)
`src/scan.c`	→ gegeben, d.h. nichts anzupassen (Infix Tokenizer Implementation)
`src/stack.h`	→ gegeben, d.h. nichts anzupassen (API UPN Werte Stack)
`src/stack.c`	→ **anzupassen**: umsetzen gemäss Angabe 2 (Implementierung)

4.1 Teilaufgabe: Stack konstruieren und wieder aufräumen

1. Führen Sie **`make test`** aus.
2. Konzentrieren Sie sich auf den ersten Test der fehlschlägt. Dies ist ein Unit Test, welcher die Funktion **`stack_new()`** prüft. Suchen Sie die Funktion in **`src/stack.h`** und **`src/stack.c`**.
   Was ist die geforderte Funktionalität und wie ist sie implementiert?

```
stack_t *stack_new(size_t max_elements)
{
    stack_t *instance = NULL;
    // 1. allocate a stack_t instance on the heap and set the instance variable to it
    // 2. call error_fatal_errno("stack_new: instance"); if failed to allocate the memory
    //    on the heap
    // 3. allocate an array of max_elements value_t's on the heap and store its address
    //    in the stack member of the stack_t instance
    // 4. call error_fatal_errno("stack_new: stack"); if failed to allocate the memory on
    //    the heap
    // 5. set the top member of the stack_t instance to the address of the "element" before
    //    the first stack array element
    // 6. set the full member of the stack_t instance to the address of the last element
    //    of the stack array
    // BEGIN-STUDENTS-TO-ADD-CODE









    // END-STUDENTS-TO-ADD-CODE
    return instance;
}
```

```
void stack_destroy(stack_t *instance)
{
    assert(instance);
    // 1. free the stack array
    // 2. free the stack_t instance
    // BEGIN-STUDENTS-TO-ADD-CODE




    // END-STUDENTS-TO-ADD-CODE
}
```

3. Führen Sie **`make test`** und korrigieren Sie obige Funktion, bis der Test nicht mehr fehlschlägt.

4.2 Teilaufgabe: restliches Stack API implementieren

Gehen Sie analog zur ersten Teilaufgabe vor:

1. Führen Sie **`make test`** aus.
2. Suchen Sie die unten aufgeführten Funktionen und implementieren Sie diese, bis die entsprechenden Test ohne Fehler durchlaufen.

```
stack_value_t stack_pop(stack_t *instance)
{
    assert(instance);
    if (stack_is_empty(instance)) error_fatal("stack_pop: empty");
    stack_value_t value;
    // 1. set the variable value to the value from the address the top member points to
    // 2. decrement by one element the address stored in the top member of the stack_t
    //    instance
    // BEGIN-STUDENTS-TO-ADD-CODE




    // END-STUDENTS-TO-ADD-CODE
    return value;
}
```

```
stack_value_t stack_top(stack_t *instance)
{
    assert(instance);
    if (stack_is_empty(instance)) error_fatal("stack_top: empty");
    stack_value_t value;
    // 1. set the variable value to the value from the address the top member points to
    // BEGIN-STUDENTS-TO-ADD-CODE




    // END-STUDENTS-TO-ADD-CODE
    return value;
}
```

```
int stack_is_empty(stack_t *instance)
{
    assert(instance);
    int is_empty = 1;
    // 1. set is_empty to 1 if the top equals the initial condition as done in
    //    stack_new(), otherwise to 0
    // BEGIN-STUDENTS-TO-ADD-CODE




    // END-STUDENTS-TO-ADD-CODE
    return is_empty;
}
```

```
int stack_is_full(stack_t *instance)
{
    assert(instance);
    int is_full = 1;
    // 1. set is_full to 1 if the top equals the full pointer as set in the stack_new()
    //    function, otherwise 0
    // BEGIN-STUDENTS-TO-ADD-CODE



    // END-STUDENTS-TO-ADD-CODE
    return is_full;
}
```

3. Führen Sie **`make test`** und korrigieren Sie, bis die beiden Tests nicht mehr fehlschlagen.

5 Aufgabe 3: Evaluator

5.1 Teilaufgabe: fehlende Operatoren implementieren

Gehen Sie analog den obigen Teilaufgaben vor und implementieren Sie, gemäss Vorgaben im Code, die fehlenden Operatoren

```
static int eval_unary(eval_t *instance, eval_op_t op)
{
    assert(instance);
    assert(instance->stack);

    unsigned int v;
    switch(op) {
    case OP_CHS:
        v = stack_pop(instance->stack);
        stack_push(instance->stack, -v);
        return 1;
    case OP_INV:
        // 1. implement the ~ operator analogous to the - sign operator above
        // BEGIN-STUDENTS-TO-ADD-CODE



        // END-STUDENTS-TO-ADD-CODE
    default:
        break;
    }
    return 0;
}
```

```
static int eval_binary(eval_t *instance, eval_op_t op)
{
    assert(instance);
    assert(instance->stack);

    unsigned int a;
    unsigned int b;

    switch(op) {
    case OP_ADD:
        b = stack_pop(instance->stack);
        a = stack_pop(instance->stack);
        stack_push(instance->stack, a + b);
        return 1;
    case OP_SUB:
        b = stack_pop(instance->stack);
        a = stack_pop(instance->stack);
        stack_push(instance->stack, a - b);
        return 1;
    case OP_MUL:
        b = stack_pop(instance->stack);
        a = stack_pop(instance->stack);
        stack_push(instance->stack, a * b);
        return 1;
    case OP_DIV:
        b = stack_pop(instance->stack);
        a = stack_pop(instance->stack);
        stack_push(instance->stack, a / b);
        return 1;
    case OP_MOD:
        b = stack_pop(instance->stack);
        a = stack_pop(instance->stack);
        stack_push(instance->stack, a % b);
        return 1;
    case OP_BIT_OR:
        // 1. implement the | operator analogous to the * operator above
        // BEGIN-STUDENTS-TO-ADD-CODE



        // END-STUDENTS-TO-ADD-CODE
    case OP_BIT_XOR:
        // 1. implement the ^ operator analogous to the * operator above
        // BEGIN-STUDENTS-TO-ADD-CODE



        // END-STUDENTS-TO-ADD-CODE
    case OP_BIT_AND:
        // 1. implement the & operator analogous to the * operator above
        // BEGIN-STUDENTS-TO-ADD-CODE



        // END-STUDENTS-TO-ADD-CODE
    case OP_BIT_LEFT:
        // 1. implement the << operator analogous to the * operator above
        // BEGIN-STUDENTS-TO-ADD-CODE



        // END-STUDENTS-TO-ADD-CODE
    case OP_BIT_RIGHT:
        // 1. implement the >> operator analogous to the * operator above
        // BEGIN-STUDENTS-TO-ADD-CODE



        // END-STUDENTS-TO-ADD-CODE
    default:
        break;
    }
    return 0;
}
```

Wenn die obigen Teilaufgaben erfolgreich umgesetzt sind, laufen die Tests ohne Fehler durch und der Integer Rechner ist voll funktionsfähig.

6 Bewertung

Die gegebenenfalls gestellten Theorieaufgaben und der funktionierende Programmcode müssen der Praktikumsbetreuung gezeigt werden. Die Lösungen müssen mündlich erklärt werden.

Aufgabe	**Kriterium**	**Gewicht**
	Sie können das funktionierende Programm inklusive funktionierende Tests demonstrieren und erklären.	
1	Papierübung Infix zu Postfix Übersetzung	1/4
2	Teilaufgabe: Stack konstruieren und wieder aufräumen	1/4
	Teilaufgabe: restliches Stack API implementieren	1/4
3	Teilaufgabe: fehlende Operatoren implementieren	1/4