Tips :)

**אם יש לכם שאלות/תהיות, באמת שתרגישו חופשיים לפרסם בקבוצה/לשלוח בפרטי, ואני אדאג לעדכן גם פה עם תשובה :)**

Please take a note, that this page will be updated in the next couple of days

Notes of Eli Kaski, he has built around 30 tests (Credited to him) :)

I will add my tests later on today and tomorrow :)  
My comments on his words, are in purple ^^  
“  
 0. START WORKING EARLY. START RIGHT NOW. this is one of the longest/hardest exercises ive ever seen

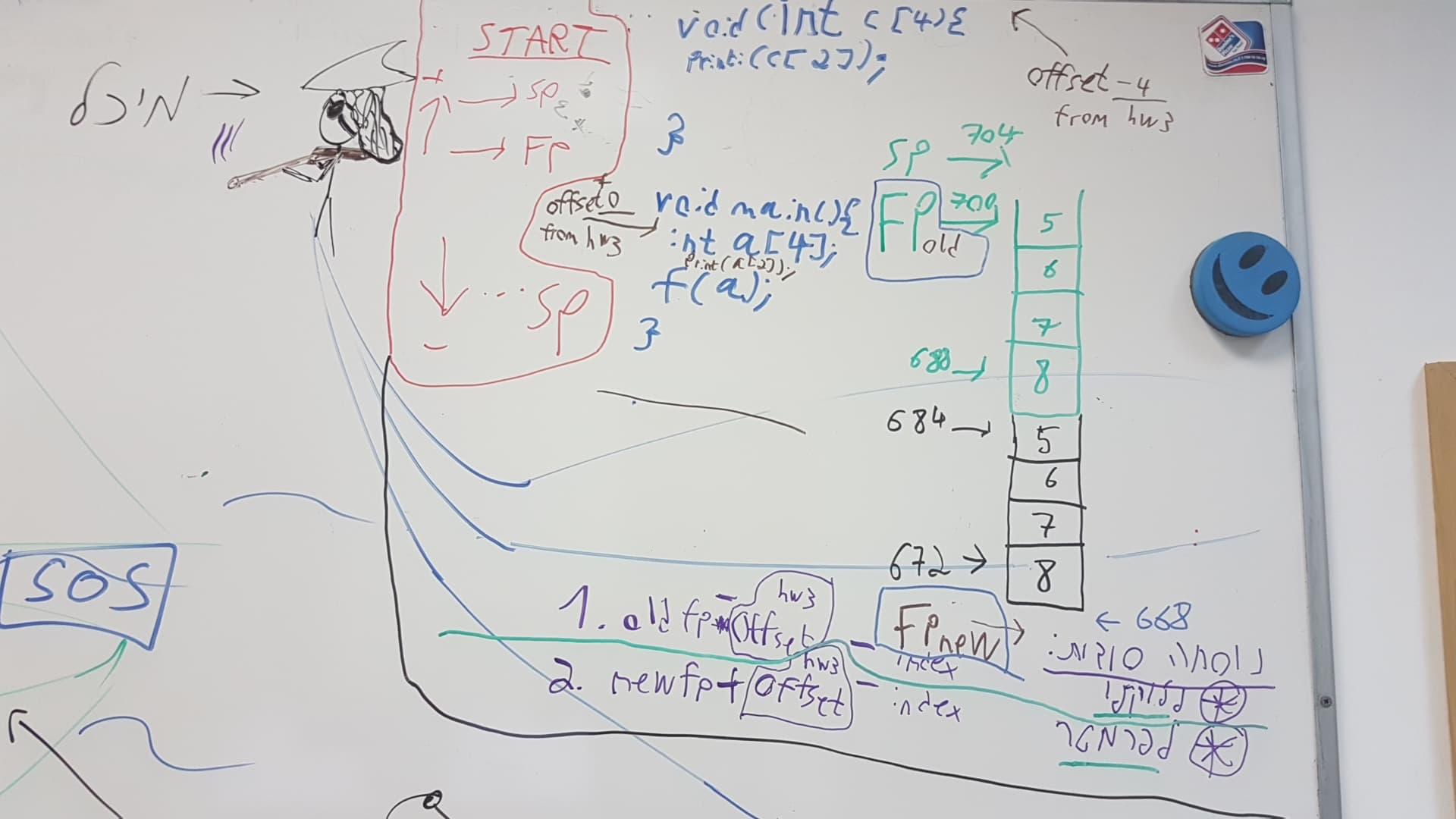
1. most important tip - make sure you know what the stack looks like. the offsets you calculated in HW3 are a tremendous help, use them! (I agree, it’s a bit confusing compared to hw3). Please look at the example below, I put there the FP (Aka frame pointer) as old and new, when we are in method main(), and then when we are in method void f(int c[4]).
2. working with arrays - understand where the array pointer points to (start or end of array), and given an index, how to calculate the address of the appropriate element. The two formulas that appear in the picture below should combat that :), as you already did most of the job anyway!! (HW3) . What we have, is that a negative offset is actually the parameters, and they are directed to higher address (NOT LOWER) and it is reverse the other way around (with local variables). Please note, that index is directed to lower address (Look at image below) ;)
3. working with bools - bools are special in this exercise. they have two "modes" - 'evaluation' (not in a register) and 'storage' (on the stack). take some time to think about how to make the transition between the two. They do seem complicated at first, because we can’t just use our registers whenever we prefer… But no worry! Kaski is right that we only use registers for booleans 2 times. First is for assigning, and second is for storing/loading from array. There is a sweet trick as to how make Booleans support the backpatching…   
   --------------------------------------  
   Consider the next case:  
   Bool z;  
   Bool x = z || z;  
   In this case, you don’t want to use in total 3 registers… as you are required to use the minimum in bools, and you can :) , **it requires just one** :) , but how? It’s a nice trick. Use a branch to prepare future bpatches :)  
   --------------------------------------
4. REGISTERS ! if you forget to return a register to the pool, YOURE GONNA HAVE A BAD TIME. i recommend on creating a function that checks whether or not you returned all the registers you took from the pool. also dont return a register before youre done using it. i learned this the hard way. You might want to add a method that reports register overflow 4 yourself :), as it will allow you to handle these cases.. But more on that in future tests.  
   “

-------------------------------------------------------------------------------------------------------------------------------

Some generals tips of my own to you :)

* **Syntax/Lexical errors of MIPS ( . il )**While you are solving this exercise, you might encounter errors of the type : ILLEGAL INSTRUCTION / A LOT OF MIPS CODE, what it usually means is that the code you emitted to the buffer/data has wrong syntax/lexical. A popular case happens when you didn’t ensure that a BOOL list was bpatched ----> beq $8, $9, \_\_\_MISSING BPATCH\_\_\_\_  
  And then you get illegal instruction :(.  
  Please, use the notepad++ MIPS tip and go over your il file :) , pasting the post from facebook here :)
* טיפ שימושי ונוסף!! :)
* לאלו מכם שרוצים לפעמים לקרוא את הקובץ il שיוצא להם (בניית הקובץ בפורמט אסמבלי (MIPS))
* הנה הדרך:
* 1- Download (File -> Download)
* [https://docs.google.com/leaf…](https://docs.google.com/leaf?id=0BweosVm_Z81ZZWJkNTNiOTItNzRkYi00ZTk3LWE0NjAtN2VkMDEyNGRiNWZh)
* 2- Open Run (Windows key+ R), type %APPDATA%/Notepad++
* 3- Paste the downloaded file in there.
* 4- Open notepad++, go to Language->Mips.
* Below link shows what will happen.
* <http://imgur.com/1fqAP>
* זה תוסף שמצאתי ל Notepad++ , שמתווסף לכם תחת Language->MIPS.
* הוא מוסיף צבעים, ובכך הקוד נעשה קריא יותר :), ומקל על ההבנה :).  
    
   **Credit to Alex Ich** on the following remark (**Color preference**):
* מי שעובד ב note++ עם רקע שחור תחליפו ב
* <styles>
* bgColor="000000"
* בכל מקום במקום :
* bgColor="FFFFF"
* **Index of array, how to do the validation (In bounds)**   
  With one register :) ---> Load index to registerIndex. Then, cmp it to register $0 , which is zero (Index is at least 0). Next, you can add **MINUS** size of array to the registerIndex. Then, compare again with $0 :) . **After** you finished the 2nd compare, please, add **PLUS** size of array to the registerIndex. You will need it again :) (To actually use array[index])
* **BPatching boolean Exp when dealing with stack(applied also to function calls)**To deal with that, we can do the following (Which is described with an example)  
  **bool b1 = b0 and true;**1. We allocate a register **before loading from stack** input 1/0 .2. We load b0 from stack..  
  3. Then, with that allocated register, we decide whether to create a true list or a false list (As register contains value of 0 or 1).   
  4. Then, we free register, as it is not needed now.   
  5. Bpatching the “and” as usual.  
  6. We allocate new register **before storing into stack** result 1/0   
  6. We have to decide our final result, with the help of the boolean lists(inside a true list or a false list, we don’t need to know which one). We bpatch them, into loading values of 1/0 respectively (TRUE / FALSE) into a new allocated register. (Meaning, true list will be bpatched to a place where we load the value of 1, and false list will be bpatched to a place where we load the value of 0)  
  7. At last, we load that register into the stack.  
  8. Now, we throw the register back into the “swimming pool” , as it’s free now :). In total, **1 allocated register**!
* **Markers M,N bring us conflicts :(**Remember that from hw3 we also had to deal with conflicts? :)  
  To solve cases of shift reduce / reduce reduce, please add your markers after tokens like ELSE / IF\_WITHOUT\_ELSE / RBRACKET or variables. The more the marker is to the right, then the bison will easily decide rather than warning about conflicts.
* **Byte,** how do we make sure that byte’s value is between 0 and 255?  
  You need to use **Mask** that was learned in Assembly 234118, to make sure you select only the lower 8 bits( 2^8 is 256 :) ).  
  **Also, use addu/subu !!!! :)**
* **Exp:** don’t use registers that are not necessary. For example,  
  The exp 3 + 7, can be done with 2 registers:  
  add $9, $9, $10  
  In general, I highly recommend you the following:  
  Each Exp that returns **a value of int/byte**, store it with a register (And assume so recursively).  
  Each Exp that returns **a boolean**, DO NOT store it. Instead, evaluate it by creating a cool trick:  
  Complete bpatch to **decide whether to load to a new register the value 1 or the value 0** :)
* **True List,** starting at Exp → true, could be in loading a stack variable / a method call result :). We store it in this basic cases, and **pass** (Mondial 2018 reference ;) ) the ball to another player, that will deal and **bpatch to the goal** destination!! :)
* **False List,** starting at Exp → false, could be in loading a stack variable / a method call result :). We store it in these basic cases, and **pass** (Mondial 2018 reference ;) ) the ball to another player, that will deal and **bpatch to the goal** destination!! :)
* **Break List**, should be dealt by the **first “grandpa”** scope that is of type while scope. So what you can do, is make sure that break list is a field of every while scope :), and let him handle the breaks!
* **Next list**, could be done in statements like If,Else and While **with a marker** :) , instead of the implementation in the lectures… Think about how you can arrange that (without letting statements have a next list). If someone still has troubles implementing.. Feel free to respond on Facebook/private.
* **Cleaning scopes:** this should be done regularly. What cases suit this description?  
  1. **General end** of a regular scope: Scopes of **While,If,Else,Other** (LBrace,RBrace)  
  There, we can allocate local fields, that after them do not exist in the method itself!! :).  
  2. **Function Scope**, we have to clean all the variables that were defined in its scope :) .  
  3. **BREAKS**: When we break, we want to clean same as any of the regular scope in 1 :)  
  4. **RETURN/RETURN EXP** When we return, we want to clean everything from current scope, up to Function scope :)
* Don’t try to build everything in one… this exercise takes a while and you can have very unexpected behaviours.
* I in my solution, had CTRL and ASM, where CTRL is the class that dealt with hw3, and ASM/MIPS is the class that was in charge of hw5. Of course… ASM did use some of CTRL’s traits, but not to change/hurt the implementation. You should avoid harming the code that worked before, as it is quite necessary!!!!!!
* I would object to using the debugger… simply because you might not find/understand and it will just frustray you even more. My plan is to build tests in levels, that way you must unlock sets of tests, to be able to be tested on the next.
* So basically, the tests get improved this time, with helpful contributors like Eli Kaski (and potential others that have some ideas :)). I decided to implement funMe withLevelOptional. You can run without LevelOptional, and it will work the same :), however if you fail in something along the way, it will report you the 1st level that you fell in.
* Level 0 is making sure that no harm was done to Hw3 compilation errors
* Level 1 is making sure that you can pass tests that even don’t have complicated requests. For example, no prints, no arrays, no while/if/else and etc… very basic at first
* Level 3, some booleans will be tested :)
* Level 4, some binops and relops will be tested :)
* While the previous levels will just make sure your code compiles, and ask you to print nothing… At some point we need to :D
* Levels 5-8 will bring prints
* Levels 9-10 will bring runtime errors
* Levels 11+ will bring arrays/functions results, compositions, serious breaks and what not
* Eli Kaski’s tests are not easy
* [**http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm**](http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm)
* **Quick Tutorial!!!!!!!! Please use it for mips commands :)**
* Many useful assembly commands:
* Using J as GOTO (Bpatches complete, here branches are dangerous if too short :( )
* Suppose you have index in register R3, and you wish to multiply it by 4 (You need to access array in respective index)  
  **Two times REPEAT  
  : add, R3, R3, R3  
  Result ---> R3 = (R3 + R3) + (R3 + R3) = 4 \* R3**
* **Don’t leave any JUMP not bpatched… it will make your spim fail :(**
* **Duplication of code is quite harmful… this exercise is huge :(**
* **Use macro for something of the kind: addi $sp, $sp, -4  
  As this case returns a lot :D , and it’s not the only one ^^**
* **Create 3 strings per error, 1. The reported error itself, 2. labelData as to where the string is defined, 3. Labael name**
* **DEFINE MARKERS M & N From hw4 :)**
* **void printScope(){**
* **/\*output::endScope();**
* **for(int i=0;i<identifiers.size();++i){**
* **output::printID(**
* **(identifiers[i].getIdName()),**
* **(identifiers[i].getOffset()),**
* **(identifiers[i].getIdType())**
* **);**
* **}\*/**
* **}**
* **Comment out your printScope, per scope!!!!!!!!! We don’t need that in this hw5 :)**
* **Struct is better than class! Meaning… we don’t need to deal in compilation with no permissions… ;)**
* **Don’t forget to clean scopes!!!!!!!!! Easy way: Rather than count how many fields, are arrays and not… and etc… :) Simply, subtract currentScope.offset to fatherScope.offset :) , it should work! :)**

An idea of how to describe array element loading, finding position using offset from hw3 and so on :)  
Will explain 2 formulas from below in the following days:

  
  
This hw is huge, tiring, troublesome and really makes you want to RUN away for some TIME :).

As this hour is a bit late and I spent mostly on making some tests here and there, I will update this page often this week.

**FAQ**

Question:

היי הילה,

"תרגיל זה מתמקד בייצור קוד אסמבלי ולא מוסיף שגיאות קומפילציה מעבר לאלה שהופיעו בתרגיל 3 .יש לדאוג שהקוד המיוצר ייטפל בשגיאת חלוקה באפס ובשגיאת חריגה ממערך שהוזכרו בפרק הסמנטיקה."

הנוסח מעט מטעה ("מעבר לאלה") ולכן מברר.

בנוגע לדיווח שגיאות של תרגיל בית 3,

האם יש להניח כי לא ייבדקו מקרים מתרגיל בית 3 (שגיאות קומפילציה)? לדוגמה:

void main(){

return 3;

}

או למשל דוגמת קובץ ריק עם אי קיום main?

או שההתנהגות הרצויה היא כמו בתרגיל הקודם:

דיווח שגיאה ויציאה מהתוכנית?

בתודה,

דניאל דימנשטיין

Answer (of TA):

**שלום דניאל,**

**הטיפול בכל שגיאות הקומפילציה של תרגיל 3 צריך להישאר בקוד שלכם. המטרה היא, בסוף תרגיל 5, לקבל קומפיילר FanC מלא, הכולל הודעה למתכנת אם הוא כתב תכנית שגויה, וייצור קוד.**

**בונוס: זה יעזור לכם לא מעט בזמן שתעבדו על התרגיל. מכיוון ששלב ייצור הקוד מניח כי קוד המקור שהוא עומד לתרגם לשפת היעד הוא קוד תקין (ולכן לא נוספות שגיאות נוספות שיש לממש בתרגיל הזה) כולנו אנשים לא מושלמים ולכתוב קוד FanC בעייתי ולקבל התנהגות לא צפויה ולבזבז עליה המון זמן זה סתם לא כיף. זו גם הסיבה שמומלץ להתחיל מלתקן את המימוש שלכם לתרגיל 3, אם יש צורך.**

**אז בקיצור - ההתנהגות הרצויה היא בדיוק כמו בתרגיל הקודם.**

**בהצלחה,**

**הילה  
----------------------------------------------**

Answer (of TA):

**Hello Daniel,**

**Order of operations including relops and unary operations (both boolean and arithmetic) is defined in HW3 in an external link. You can see the expected parse tree, and therefore the expected result, there.**

**Good luck,**

**Hila**

Follow up:

**Updating the tests + posting this link so the relevant association can be seen :)**

[**https://introcs.cs.princeton.edu/java/11precedence/**](https://introcs.cs.princeton.edu/java/11precedence/)