DKK compiler project 2025-2026

Konstantinos Oikonomopoulos, Dimosthenis Zempilas, Kostantin Ziu

February 11, 2026

Contents

1 Introduction

1.1 Authors

Konstantinos Oikonomopoulos, AEM:04012, koikonomop@uth.gr,
Dimosthenis Zempilas, AEM:03857, dzempilas@uth.gr,
Konstantin Ziu, AEM:03950, kziou@uth.gr

1.2 README

This compiler, the "dkk compiler", named after our first name initials, does syntactical and semantic analysis, assisted by lectical analysis, providing Intermediate Representation and some largely unverified RISC-V assembly code, for a C++ inspired language "CPP". We apologize for the current state of the repository however we can proudly say that all provided code is AI-free and completely human written.

This report unfortunately has been written after the partial completion of the compiler, and the authors find it almost impossible to discuss all nooks and crannies of the code and its contents. The explanation of the semantic analysis of the compiler automaton will be consequently undermined by the most challenging part of the production of the final RISC-V assembly representation. The authors pledge to answer any questions that might persist after reading this report, we are free to contact through our emails koikonomop@uth.gr, dzempilas@uth.gr, kziou@uth.gr.

1.3 Preface before the Demonstration

The best way to present the compiler is to demonstrate with some input examples showcasing the outputs. However we must preface the demonstration with a general description of the code. Focusing on the syntax and semantics of the analysis done in bison, we use a grammar parser to appropriately determine the reduction of the coding language. The symbol table used is a hash-table that stores everything declared in a scope. A node of the symbol table contains the name, datatype and scope as well as the array information, the dimensions and the size of the dimensions, and the class information such as the members and methods, the superclass and also the visibility. Many checks are made to verify the semantics of the input, such as comparing the types, the sizes of the arrays indexed and the correct dimensions, the existence of variables referenced, the custom types, etc. . Finally for the production of the IR and the assembly, keywords ("while", "for", "if") are given meaning and all data declared is formatted and stored in the Binding table that is in practice a list that stores the contents declared inside functions and classes as well as their necessary information and offsets.

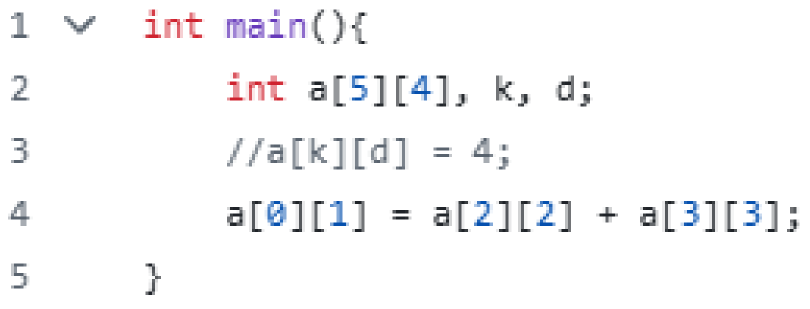
1.4 What is missing and Future works

The number one concern of the authors is verification, for such a job an emulator could be used to compare the correctness of the compilers output.
The semantic analysis is complete for all cases concerning the requirements of the class project for the most part, however there are many cases where a semantic error results in segmentation faults instead of an error message, for the future works we are aiming to solve all such cases.
Furthermore there are a few concerns regarding the production of the final assembly representation. The first one is the fact that there was a change from the original syntax provided; The exact change was the removal of initialization of static variables with non constants. Similarly, no assembly code is provided for lists and the addition of lists. This is because we believe that it requires run-time functionality or very advanced semantic analysis.
Finally one of the many issues that remain is the fact that as it is currently written, function parameters are passed through parameter registers ("$a0, $a1, etc") and consequently allow only for 32bit types to be stored in them. Another known issue is that loading and storing from the memory is done only in 4byte words and there is no evaluation or management of the size of the values loaded and stored.

2 Demonstration

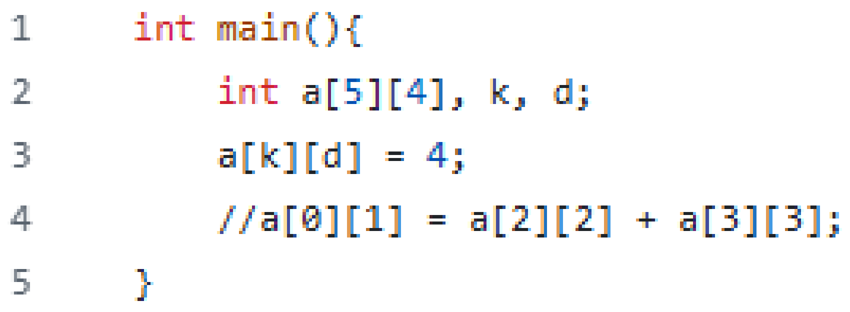
2.1 Array example

Let's begin by demonstrating with some array examples. Here is an integer array that has is indexed, inbounds and out of bounds by constants and variables. Only the value of the constants is calculated to determine whether or not the array is indexed correctly. The error messages and the IR and RISC-V assembly outputs are shown below.



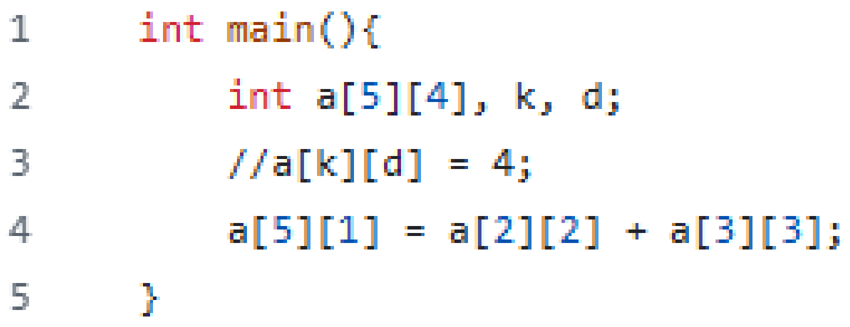
```
int main(){
    int a[5][4], k, d;
    //a[k][d] = 4;
    a[0][1] = a[2][2] + a[3][3];
}
```

Figure 1: Correct constant indexing



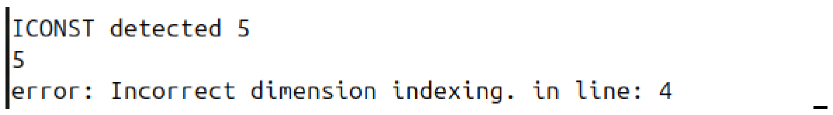
```
1   int main(){
2       int a[5][4], k, d;
3       a[k][d] = 4;
4       //a[0][1] = a[2][2] + a[3][3];
5   }
```

Figure 2: Correct variable indexing



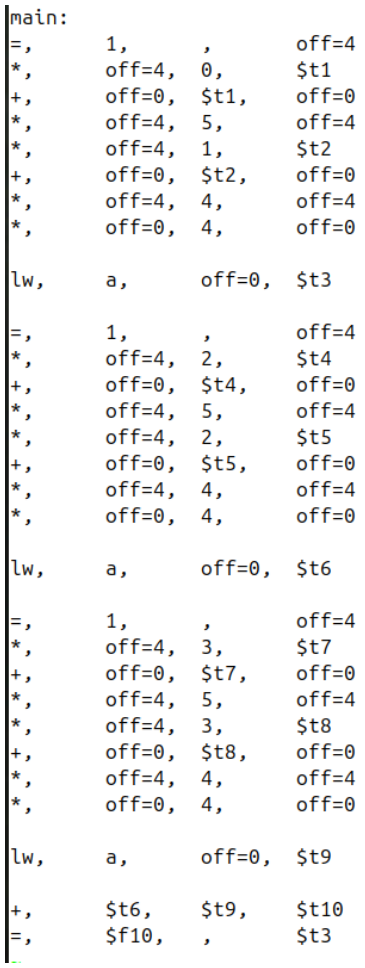
```
1   int main(){
2       int a[5][4], k, d;
3       //a[k][d] = 4;
4       a[5][1] = a[2][2] + a[3][3];
5   }
```

Figure 3: Incorrect constant indexing



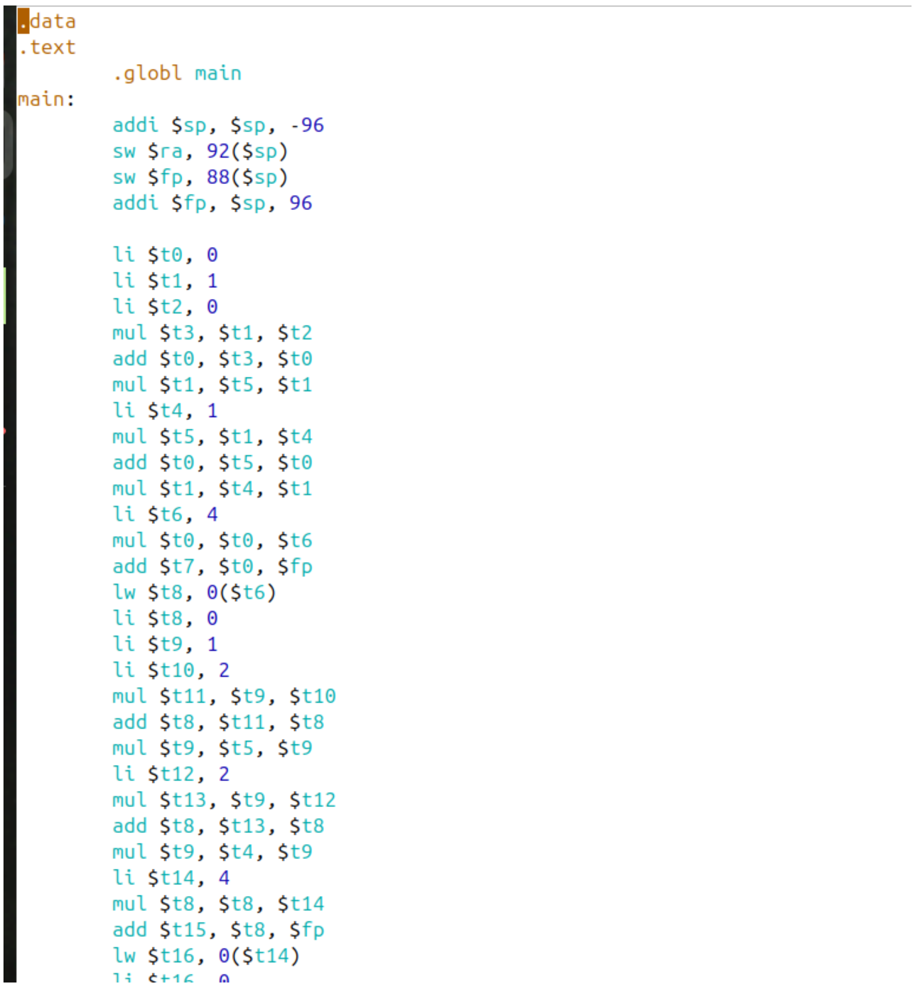
```
ICONST detected 5
5
error: Incorrect dimension indexing. in line: 4
```

Figure 4: Error from incorrect indexing



```
main:
=,      1,      ,       off=4
*,      off=4,  0,      $t1
+,      off=0,  $t1,    off=0
*,      off=4,  5,      off=4
*,      off=4,  1,      $t2
+,      off=0,  $t2,    off=0
*,      off=4,  4,      off=4
*,      off=0,  4,      off=0

lw,     a,      off=0,  $t3

=,      1,      ,       off=4
*,      off=4,  2,      $t4
+,      off=0,  $t4,    off=0
*,      off=4,  5,      off=4
*,      off=4,  2,      $t5
+,      off=0,  $t5,    off=0
*,      off=4,  4,      off=4
*,      off=0,  4,      off=0

lw,     a,      off=0,  $t6

=,      1,      ,       off=4
*,      off=4,  3,      $t7
+,      off=0,  $t7,    off=0
*,      off=4,  5,      off=4
*,      off=4,  3,      $t8
+,      off=0,  $t8,    off=0
*,      off=4,  4,      off=4
*,      off=0,  4,      off=0

lw,     a,      off=0,  $t9

+,      $t6,    $t9,    $t10
=,      $f10,   ,       $t3
```

Figure 5: Intermediate representation



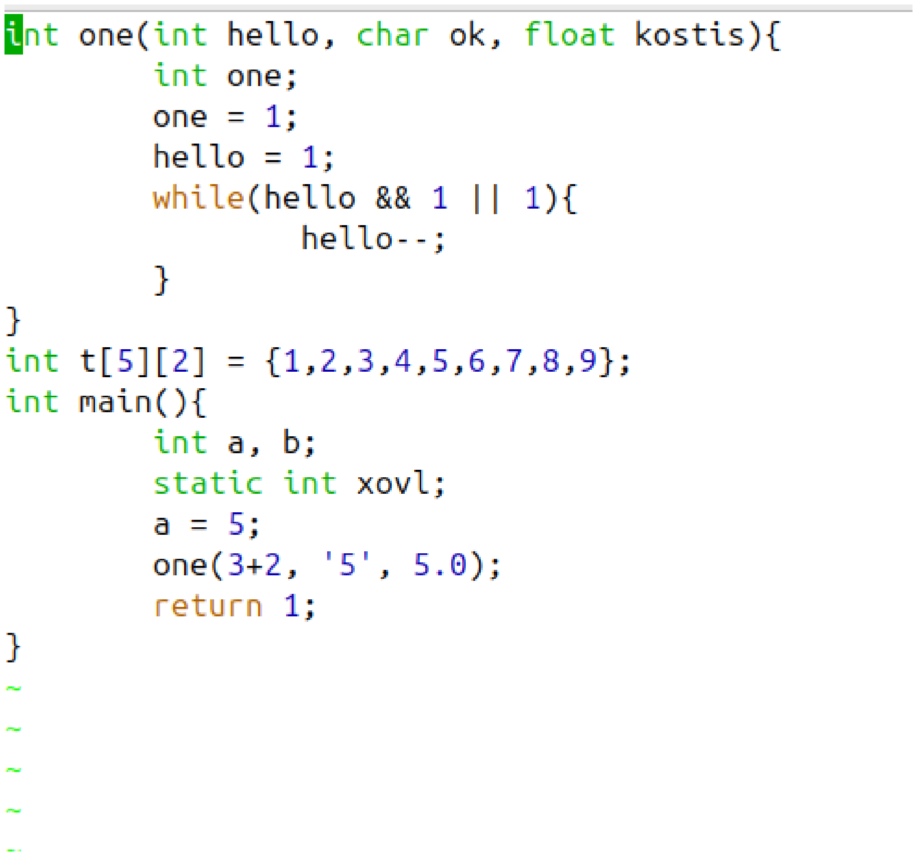
```
.data
.text
        .globl main
main:
        addi $sp, $sp, -96
        sw $ra, 92($sp)
        sw $fp, 88($sp)
        addi $fp, $sp, 96

        li $t0, 0
        li $t1, 1
        li $t2, 0
        mul $t3, $t1, $t2
        add $t0, $t3, $t0
        mul $t1, $t5, $t1
        li $t4, 1
        mul $t5, $t1, $t4
        add $t0, $t5, $t0
        mul $t1, $t4, $t1
        li $t6, 4
        mul $t0, $t0, $t6
        add $t7, $t0, $fp
        lw $t8, 0($t6)
        li $t8, 0
        li $t9, 1
        li $t10, 2
        mul $t11, $t9, $t10
        add $t8, $t11, $t8
        mul $t9, $t5, $t9
        li $t12, 2
        mul $t13, $t9, $t12
        add $t8, $t13, $t8
        mul $t9, $t4, $t9
        li $t14, 4
        mul $t8, $t8, $t14
        add $t15, $t8, $fp
        lw $t16, 0($t14)
        li $t16, 0
```

Figure 6: Part of assembly output

There is so much to be said about the analysis of the semantics and the production of the IR and assembly. In sort, for the semantic part, the dimensions and their sizes are saved in the symbol table, with a stride algorithm, the indexing is calculated as an offset for the arrays. After making sure that there are no errors, for the IR the calculation of the indexes is translated to a "quadruples" representation where the stride is represented as "off=4" and the index as "off=0".
After calculating the final index a pseudo "lw" instruction states that from the variable "a" should be loaded with an offset of "off=0" (the index) into the register $tx.
Much like the IR, the array index is calculated in the same way for the assembly. The size of the stack memory referenced is calculated "a posteriori" and placed above the functional part of the code.

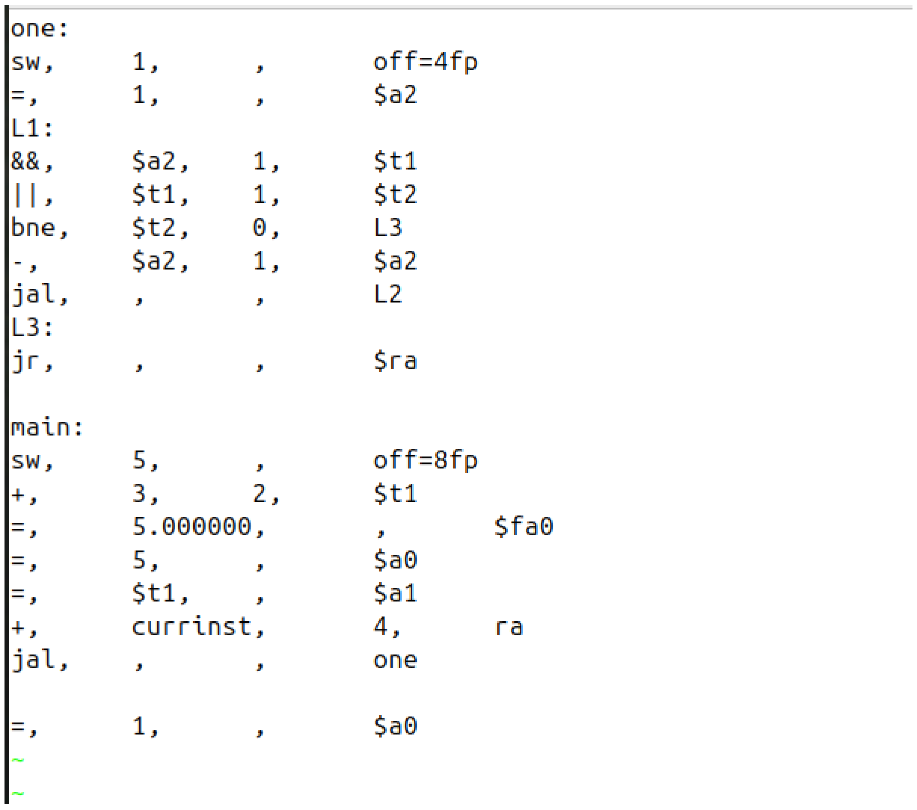
2.2 Advanced example

Here's a more advanced example where we deal with 1. functions, 2. function parameters, 3. static declaration, 4. multiple declarations in one line, 5. while statement 6. logical backpatching, 7. INCDEC backpatching, 8. float RegFile conversion



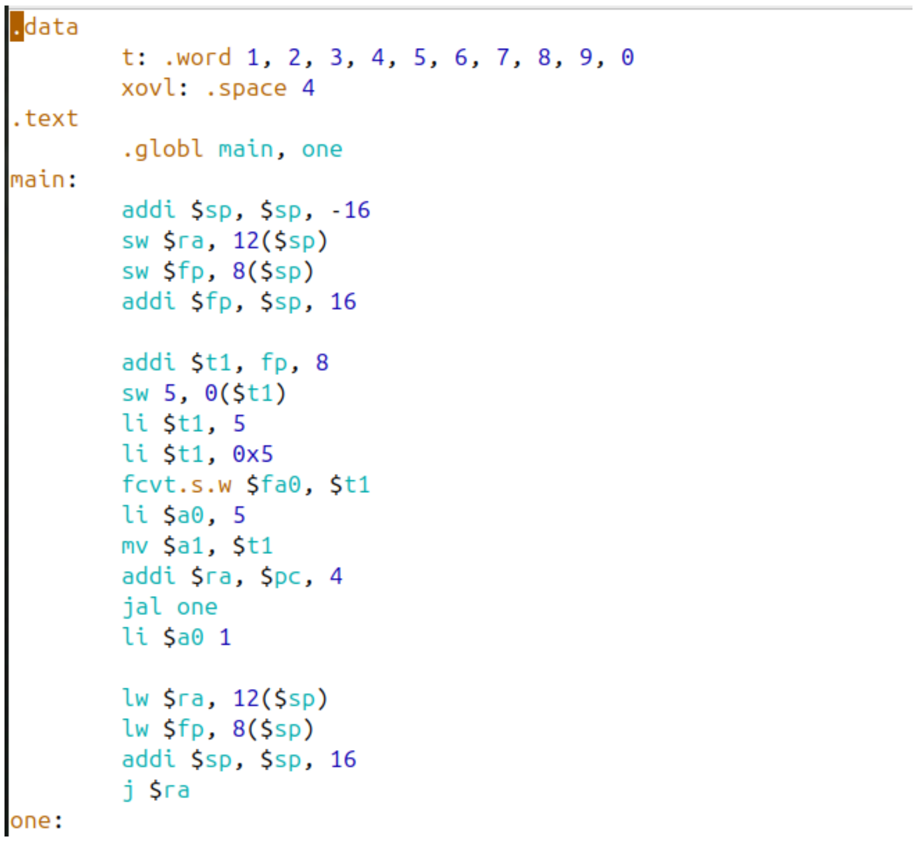
```
int one(int hello, char ok, float kostis){
        int one;
        one = 1;
        hello = 1;
        while(hello && 1 || 1){
                hello--;
        }
}
int t[5][2] = {1,2,3,4,5,6,7,8,9};
int main(){
        int a, b;
        static int xovl;
        a = 5;
        one(3+2, '5', 5.0);
        return 1;
}
~
~
~
~
~
```

Figure 7: CPP code example



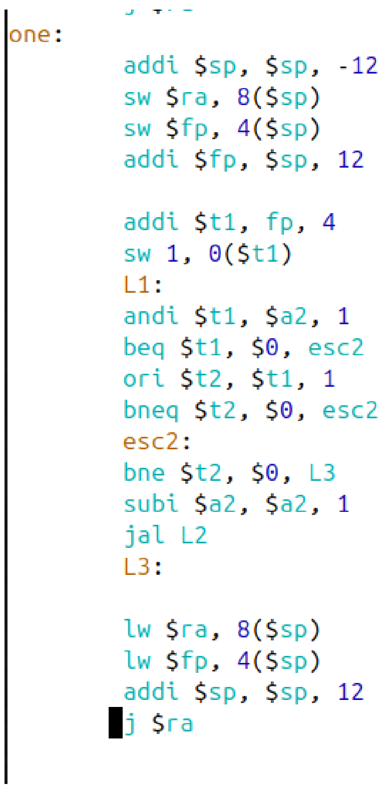
```
one:
sw,     1,      ,       off=4fp
=,      1,      ,       $a2
L1:
&&,     $a2,    1,      $t1
||,     $t1,    1,      $t2
bne,    $t2,    0,      L3
-,      $a2,    1,      $a2
jal,    ,       ,       L2
L3:
jr,     ,       ,       $ra

main:
sw,     5,      ,       off=8fp
+,      3,      2,      $t1
=,      5.000000,       ,       $fa0
=,      5,      ,       $a0
=,      $t1,    ,       $a1
+,      currinst,       4,      ra
jal,    ,       ,       one

=,      1,      ,       $a0
~
~
```

Figure 8: IR output



```
.data
        t: .word 1, 2, 3, 4, 5, 6, 7, 8, 9, 0
        xovl: .space 4
.text
        .globl main, one
main:
        addi $sp, $sp, -16
        sw $ra, 12($sp)
        sw $fp, 8($sp)
        addi $fp, $sp, 16

        addi $t1, fp, 8
        sw 5, 0($t1)
        li $t1, 5
        li $t1, 0x5
        fcvt.s.w $fa0, $t1
        li $a0, 5
        mv $a1, $t1
        addi $ra, $pc, 4
        jal one
        li $a0 1

        lw $ra, 12($sp)
        lw $fp, 8($sp)
        addi $sp, $sp, 16
        j $ra
one:
```

Figure 9: RISC-V assembly output 1

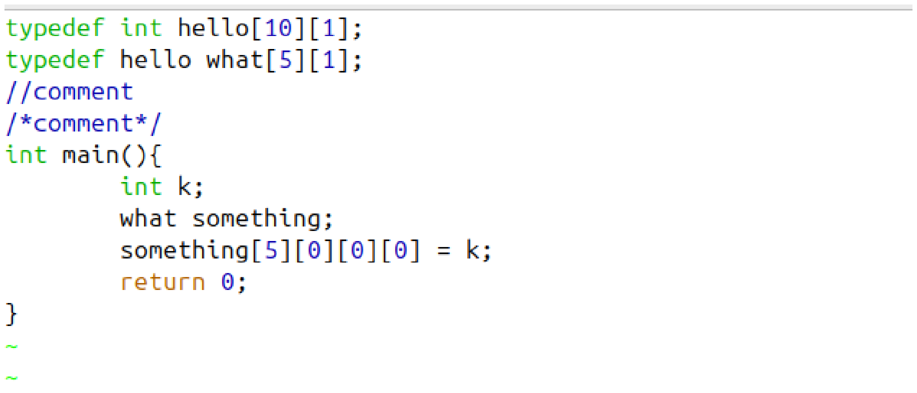


```
one:
        addi $sp, $sp, -12
        sw $ra, 8($sp)
        sw $fp, 4($sp)
        addi $fp, $sp, 12

        addi $t1, fp, 4
        sw 1, 0($t1)
        L1:
        andi $t1, $a2, 1
        beq $t1, $0, esc2
        ori $t2, $t1, 1
        bneq $t2, $0, esc2
        esc2:
        bne $t2, $0, L3
        subi $a2, $a2, 1
        jal L2
        L3:

        lw $ra, 8($sp)
        lw $fp, 4($sp)
        addi $sp, $sp, 12
        j $ra
```

Figure 10: RISC-V assembly output 2

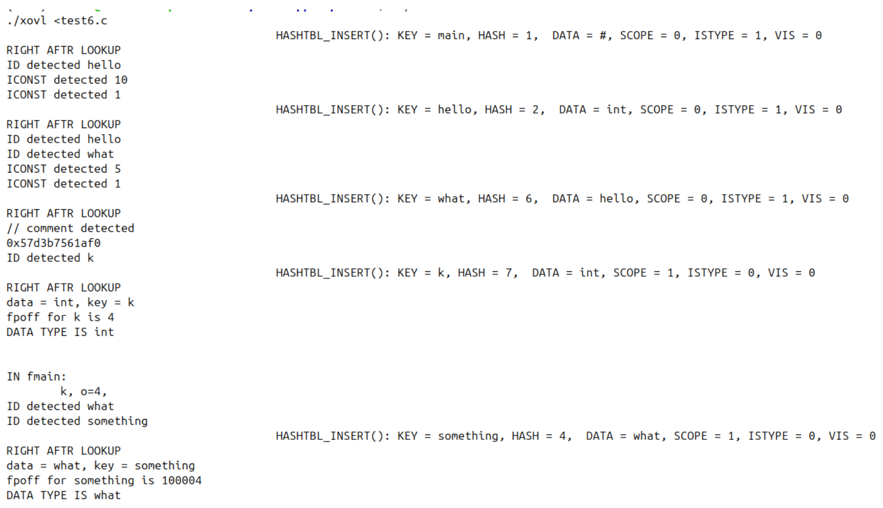
Looking at the assembly file output the first thing we notice is that the variable ”t” is initialized and the missing values get assigned to 0. The static variable declared in main is also included in the global memory. Moving downwards we notice that the stack is 16bytes or 2 words more than the 8bytes of the $fp and $ra, corresponding to variables declared inside the function main. To use the floating point value we convert with the RISC-V convert instruction ”fcvt.s.w” for single precision from integer to FP.
Inside the function ”one” we use the parameter registers instead of the stack. The ”or” and ”and” instruction lead to branch instructions in order to respect the evaluation sequence, and the ”while” statement needs the labels ”L1” and ”L3”.

3 Some more examples



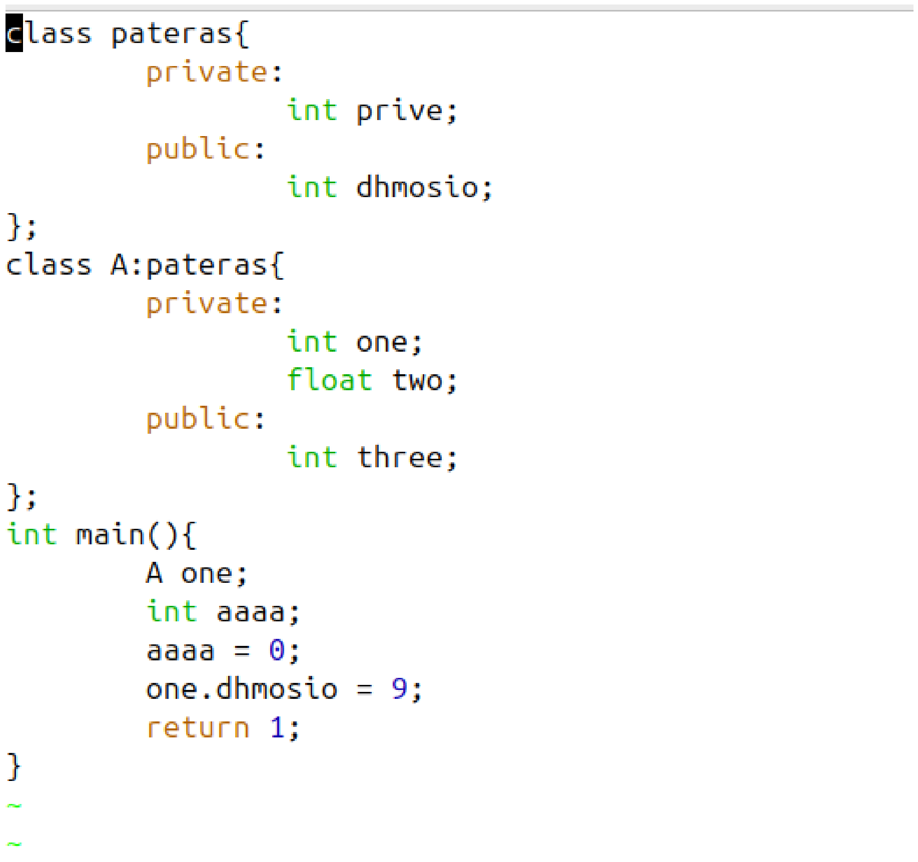
```
typedef int hello[10][1];
typedef hello what[5][1];
//comment
/*comment*/
int main(){
        int k;
        what something;
        something[5][0][0][0] = k;
        return 0;
}
~
~
```

Figure 11: typedef



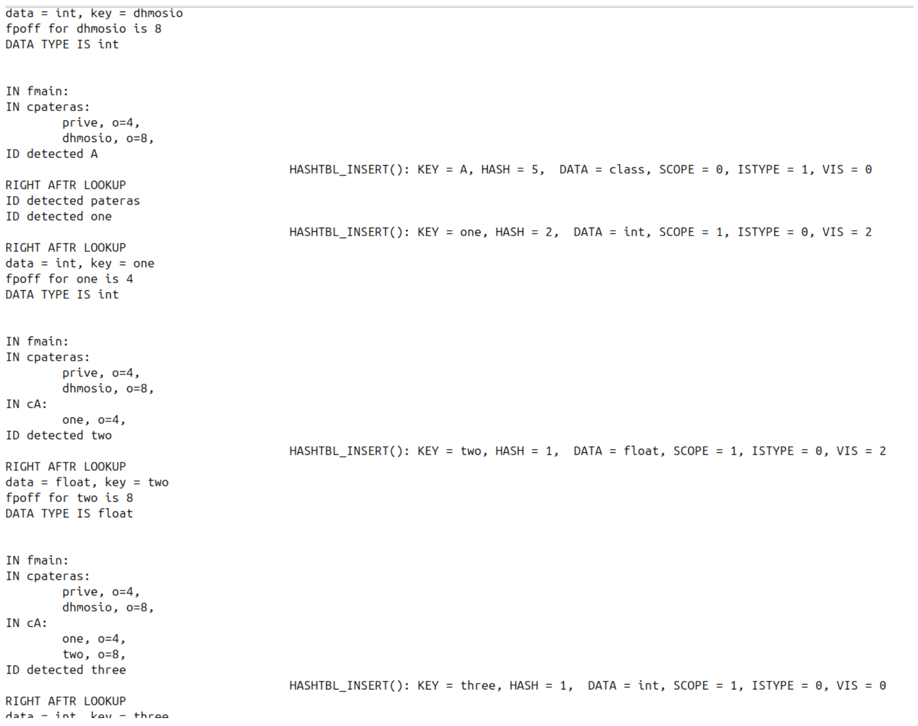
```
./xovl <test6.c
                                        HASHTBL_INSERT(): KEY = main, HASH = 1,  DATA = #, SCOPE = 0, ISTYPE = 1, VIS = 0
RIGHT AFTR LOOKUP
ID detected hello
ICONST detected 10
ICONST detected 1
                                        HASHTBL_INSERT(): KEY = hello, HASH = 2,  DATA = int, SCOPE = 0, ISTYPE = 1, VIS = 0
RIGHT AFTR LOOKUP
ID detected hello
ID detected what
ICONST detected 5
ICONST detected 1
                                        HASHTBL_INSERT(): KEY = what, HASH = 6,  DATA = hello, SCOPE = 0, ISTYPE = 1, VIS = 0
RIGHT AFTR LOOKUP
// comment detected
0x57d3b7561af0
ID detected k
                                        HASHTBL_INSERT(): KEY = k, HASH = 7,  DATA = int, SCOPE = 1, ISTYPE = 0, VIS = 0
RIGHT AFTR LOOKUP
data = int, key = k
fpoff for k is 4
DATA TYPE IS int


IN fmain:
        k, o=4,
ID detected what
ID detected something
                                        HASHTBL_INSERT(): KEY = something, HASH = 4,  DATA = what, SCOPE = 1, ISTYPE = 0, VIS = 0
RIGHT AFTR LOOKUP
data = what, key = something
fpoff for something is 100004
DATA TYPE IS what
```

Figure 12: typedef output



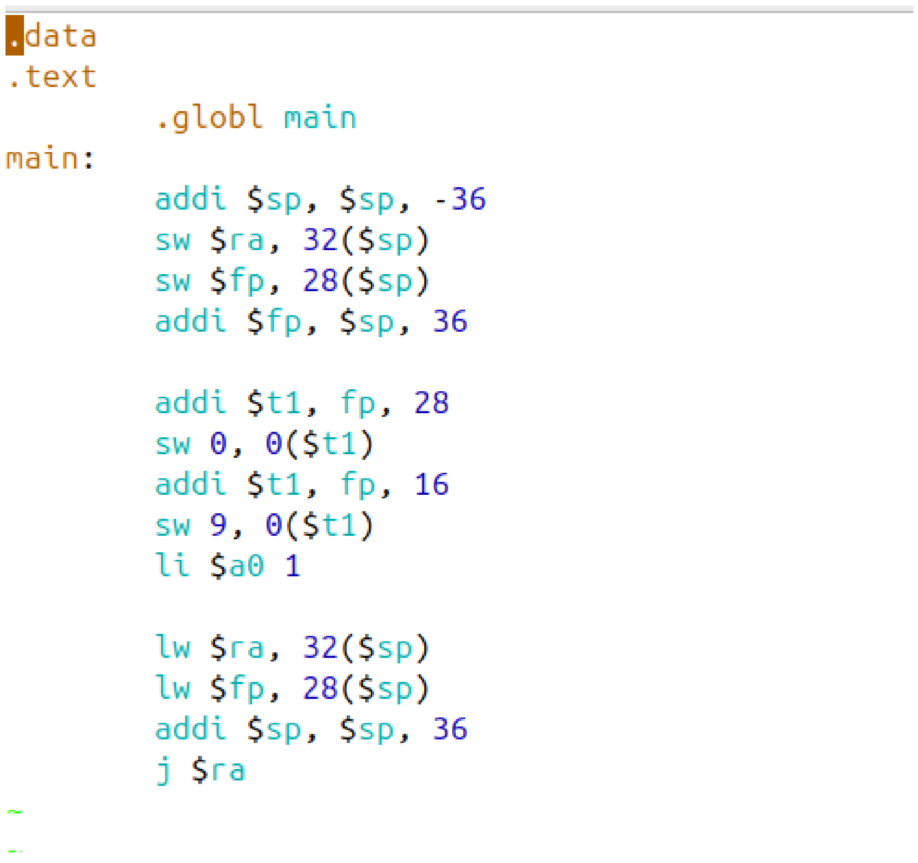
```
class pateras{
        private:
                int prive;
        public:
                int dhmosio;
};
class A:pateras{
        private:
                int one;
                float two;
        public:
                int three;
};
int main(){
        A one;
        int aaaa;
        aaaa = 0;
        one.dhmosio = 9;
        return 1;
}
~
~
```

Figure 13: Classes example



```
data = int, key = dhmosio
fpoff for dhmosio is 8
DATA TYPE IS int


IN fmain:
IN cpateras:
        prive, o=4,
        dhmosio, o=8,
ID detected A
                                        HASHTBL_INSERT(): KEY = A, HASH = 5,  DATA = class, SCOPE = 0, ISTYPE = 1, VIS = 0
RIGHT AFTR LOOKUP
ID detected pateras
ID detected one
                                        HASHTBL_INSERT(): KEY = one, HASH = 2,  DATA = int, SCOPE = 1, ISTYPE = 0, VIS = 2
RIGHT AFTR LOOKUP
data = int, key = one
fpoff for one is 4
DATA TYPE IS int


IN fmain:
IN cpateras:
        prive, o=4,
        dhmosio, o=8,
IN cA:
        one, o=4,
ID detected two
                                        HASHTBL_INSERT(): KEY = two, HASH = 1,  DATA = float, SCOPE = 1, ISTYPE = 0, VIS = 2
RIGHT AFTR LOOKUP
data = float, key = two
fpoff for two is 8
DATA TYPE IS float


IN fmain:
IN cpateras:
        prive, o=4,
        dhmosio, o=8,
IN cA:
        one, o=4,
        two, o=8,
ID detected three
                                        HASHTBL_INSERT(): KEY = three, HASH = 1,  DATA = int, SCOPE = 1, ISTYPE = 0, VIS = 0
RIGHT AFTR LOOKUP
data = int, key = three
```

Figure 14: Classes example output



```
.data
.text
        .globl main
main:
        addi $sp, $sp, -36
        sw $ra, 32($sp)
        sw $fp, 28($sp)
        addi $fp, $sp, 36

        addi $t1, fp, 28
        sw 0, 0($t1)
        addi $t1, fp, 16
        sw 9, 0($t1)
        li $a0 1

        lw $ra, 32($sp)
        lw $fp, 28($sp)
        addi $sp, $sp, 36
        j $ra
~
~
```

Figure 15: Classes example assembly
(and why OOP is so counterproductive memory wise)