Project Milestone 3

Kevin Ayuque (kja306), Goktug Saatcioglu (gs2417)

May 2019

1 Introduction

This is the documentation for Milestone 3 of the Spring 2019 Compiler Construction course. We had worked together on Milestones 1 and 2 and continue working together for Milestone 3. The translation logic and schemes used is given in Section 2. Our explanation of assumptions and choices made is in Section 3. Section 4 describes our register allocation scheme which is then followed up by Section 5 which details how we used the attributes given to us and the attributes we defined ourselves. Section 6 is our work log. Finally, Section 7 includes some tests and our own custom tests showing our compiler in action. Other than this file we are submitting a HACS file called `Pr3GoktugKevin.hx` that has our working compiler, the folder `/samples/` that has the test files given out with milestone 1 and the folder `/custom_samples/` that has our own test cases. To compile the compiler, navigate to the folder containing `Pr3GoktugKevin.hx` and run the following command:

```
> $HOME/.hacs/bin/hacs Pr3GoktugKevin.hx
```

To run a custom test case execute the following command:

```
> ./Pr3GoktugKevin.run --sort=Compile custom_samples/FILE_NAME.EXT
```

To run a test case from Milestone 1 execute the following command:

```
> ./Pr3GoktugKevin.run --sort=Program samples/FILE_NAME.EXT
```

All commands should be executed in a terminal session. We are submitting the same code and same documentation for both of us where the log in Section 6 indicates what was done by whom.

2 Translation Logic and Schemes

We translate MiniC into MinArm32 using various recursive schemes that start from the top level of the tree, i.e. from statements, and recursively translate smaller sub-expressions until terminating at the base cases of the statements and expressions grammars. Below we list each scheme declared and how they work.

- Scheme `S`: The scheme `S` works on multiple statements (compared to `SSingle` which works on only single statements). It's primary purpose is to pass information from one statement to the rest of the statements. When a variable is declared we update the `vt` table and store the information regarding the variable's relative position to the current stack frame. Similarly, when a pointer variable is declared we update both `vt` and `vt2` so that if we ever try to do pointer arithmetic we can detect whether the declared variable was a pointer or not. For conditional statements we delegate the task of checking for the condition to the scheme `ECond` which will be explained later. In if conditionals we use another scheme `IT` to translate the statement that come after the if. For while statements we use `SSingle` to translate the body of the while statement accordingly. The main role of `S` for conditionals is to pass labelling information to each translation call and label the translation so as to ensure that the branch statements obtained in the other function calls go to the correct places. Next, we have assignment statements and return statements and these are handled accordingly using the `AE` and `RE` schemes respectively. The primary reason we delegate to other schemes is to make the code more modular and secondarily we would like to do register allocation at the sub-scheme level because we have found that to be easier. Finally, the overarching goal of the `S` scheme is to pass all attributes from one statement to another and recursively translate all statements.

- Scheme `SingleS`: The scheme `SingleS` serves the same purposes as `S` but only translates productions that do not create more statement afterwards. The reason this scheme is necessary is because some productions only derive a single statement which in turn may or may not derive more statements. So we have to cover all cases of the derivations of the MiniC grammar otherwise we will not be able to correctly translate all valid MiniC programs. As an added bonus, we get some efficiency in the case of variable declarations as if a variable is declared and no other statements follow the variable declaration this variable can be never used. So we choose to never allocate the statement on the stack which is a small but nice addition to the translator.

- Scheme `IT`: Next, we have the scheme `IT` which translates the tail part of if statements. It is split into two cases where the first is if no tail except a statement is derived. In this case if the test is successful we branch to the true label and otherwise we branch to the false label which is immediately followed by the next label so as to fall through into the rest of the program. The branching job is delegated to the scheme `ECond`. In the other case where there is an else branch then the logic is similar but now the false label actually goes to other code. The primary job of the `IT` scheme is to correctly label the program and call the right schemes to translate the smaller sub-expressions.

- Scheme `AE`: The primary purpose of the `AE` is to carry out a job that actually belongs to the type checker. In MiniC we can only make assignments to l-values so must make sure to avoid non-sensicals assignments such as `5 + 5 = x;`. Note that such assignments can be derived from the grammar but we must reject them. Thus, the `AE` scheme checks if the left side is an l-value and then makes the relevant assignment. However, we are not exactly correct as it is still possible to do an assignment such as `**5 = x;`. It is clear that this job is better suited for the typechecker but we try to do it here. Overall, there may be undefined behavior in compiled programs if no type checking is done. Finally, this part also does some register allocation which will be discussed later on.

- Scheme `RE`: This scheme computes the result of the expression we wish to return and then places inside register 0 as per the MinArm32 specification. It also tells the sub-expression what registers it can use so that we obtain correct behavior.

- Schemes `FC` and `FCTail`: These two schemes translate function calls. The scheme `FC` kicks of the translation by translating either the first argument of the function or if the function doesn't take any arguments then it calls the function. If the function has any more arguments then we call `FCTail` to translate more of the arguments and store each one in the relevant registers. Note that `FCTail` takes a second argument which eliminates each register for a function call until we run out of registers. If we were to have a function call that has more than 4 arguments we throw an error and stop the compilation process. The reason for this is that the MinArm32 specification states that functions can take more than 4 arguments but the starter implementation given to use throws an error if a function declaration has more than 4 arguments so if there are more arguments in a function call we must exit the compilation process. As above, we do register allocation which we will explain later on. Finally, `FC` makes sure to save the current values of registers 0 through 3 on the stack and then restore them once the function call is completed. This is necessary as arguments are only stored in registers so we shouldn't lose any arguments if we call another function within a function. We assume that the way a function return is handled is correct and since we do not touch register 12 we are also sure we are not breaking the code generated by the skeleton implementation provided to us.

- Schemes `E` and `ECond`: Finally, we have these two schemes that translate the smaller sub-expressions of some expression. `ECond` considers what the conditional we are checking for is and if it is a unary operator saves one register for the result and if it is a binary operator it saves two registers for the result. Then `ECond` makes the comparison and makes the relevant branches if necessary. Branching information is provided by the attributes `true` and `false`. In a similar manner, `E` recursively translates sub-expressions and stores results in the first free register it was given. If a function call is being made we call that function and if pointer arithmetic is being done we can also achieve this. Furthermore, since the semantics of tests are well-defined we can actually evaluate an expression such as `0 > 1` to `0` for false and `1 > 0` to `1` for true so we can handle such expressions too. Also, we translate memory loads properly and if we ever encounter a string or an integer constant we store the result in memory using `DCI/DCS` and then load the result into a free register.

We also have the following utility functions that perform some helpful tasks.

- Scheme `Decr`: This scheme takes an offset and adds four to it. So we assume the stack is increasing upward as per the MinArm32 specification so we calculate the offset of local variables using this scheme.
- Scheme `Str` and `StrR`: This scheme makes a memory store or register copy depending on the context.
- Scheme `Ldr`: This scheme make a memory load into a specific memory location.

3 Assumptions and Choices Made

We need to make various assumptions due to both ambiguities and the incompleteness of the assignment paper. They are as listed below.

- Even though we check for whether a variable has been declared we do not do any further type checking related tasks. So there is a whole list of possible issues such as down-casting an int to a char or assigning an integer other than 0 to a pointer. This is undefined behavior in C but we do not know what the semantics of MiniC are. Furthermore, we do not check whether a function has been declared or not when a call is being made as this is impossible to do so within the current code given to us. This is because we declare identifiers to be a HACS `symbol` so we can only pattern match on them if they show up in the program text. In that case, we cannot make calls to library functions specified in the MinArm32 specification which leads to incorrect code. Of course, we also generate incorrect code in our current implementation but at least a type checker can catch these errors rather than us not generating any code at all. We could have modified the grammar to make this work but we did not want to break the base code. Additionally, we do not type check whether if an actual argument passed into a function matches the type of the formal argument. So overall, we do not type check the program and assume that at some point type checking will be done so that we do not end up with unsafe code.
- Another reason why we do not check whether a function has been declared is because of the library function specified in the MinArm32 specification. Instead we assume we have some sort of linker that will load the library functions onto our generated code. Of course, the linker should also issue a warning if we try to use an undefined function but this is not really the job of linking. Thus, the addition of a type checker is a must. Overall, we assume that library functions will be given to the compiler at some point so that the calls can be made properly.
- We assume all pointers are 4 bytes as MinArm32 is a 32-bit architecture. Thus, this is a simple but correct assumption to make even though it is not stated in the assignment specification.
- Whenever we use a constant integer we write it into memory and then load it from there. The reason for this is because in MinArm32 a constant `#` can only be between 0 and 255. So rather than checking for this range we simply store the number in memory using DCI and then load it from there. This in inefficient but the code is correct so we decided to emphasize correctness over efficiency.
- We assume that string comparisons such as `''hello'' < ''world''` is undefined behavior. We generate some code for this but we the behavior such a program will perform if we were to run the generated code will cause undefined behavior. We have other bits of undefined behavior due to type casts and unchecked expression so we also make a strong assumption that we will be given "correct" code (which is a vague term but makes sense in this context). If we are not given correct code we know that our generated MinArm32 will have undefined behavior and may fail or do something else.
- We assume that we can only get the address of variables. This makes sense and we stuck to C semantics to make this decision. So it is impossible to generate code for `&5` but we can generate code for `&x` which is correct behavior. Of course if a variable has not been defined then we throw an exception and stop compiling.
- We do not support functions with more than 4 arguments which has also been stated above.

- We allow for statements like `***5 = 5;` as we do not type check so assume that such expressions are passed to our compiler. However, we can correctly translate expressions such as `*x = 5;` so in terms of dereferenced assignments we generate all correct cases but we also generate incorrect cases if given to us. On the other hand, assignments to variables are correctly handled.
- We found no particular use for the strings are zero-terminated semantics of MiniC but we do not our implementation breaks this. As long as DCS appends a 0 at the end of a stored string our implementation will be correct.
- We successfully get correct semantics for tests and include tests such as `if (x)` where the conditional evaluates to true as long as `x` is not 0. We go a step further and actually allow assignment such as `x = 0 < 1;` which will assign 1 to `x` as the condition is true. While this is not specified in the MiniC semantics we allowed for this kind of evaluations as it makes sense us and adds expressivity to the language. Such assignments are also allowed in C so we are also closer to C semantics which is good to do.
- Another job of a type checker is to check whether we are always returning something and whether the type we are returning matches up with the declared return type. As stated numerous times, we do not do type checking so we also assume that when we are given a program there should be something returned.
- Note that while we made many assumptions regarding typing which have been outlined above, some of these assumptions are not exactly necessary. For example, if a given function doesn't have a return statement then we can just use whatever value was in register 0. Similarly, if a function takes 2 arguments but we called it with 4 we can just use the first two register and discard the next two. If a function takes 3 arguments but we gave it 2 then we call it with whatever default value was in $R2$. Note that these are all instances of undefined behavior so we had to make the assumptions given below. We could of course relax our assumptions and allow for undefined behavior so that our program gets a more relaxed semantics. The choice between the assumption semantics and the undefined behavior semantics is up to the user (with the latter leading a much simpler type checker/loader).

4 Register Allocation Scheme

We implement the naive register allocation strategy from the homework assignment. So every time there is a binary operator we allocate one of the registers in unused to the result of the left operand and allocate another register in unused to the result of the right operand. Then we pass this information down to sub-expressions letting them know that they can use only a specific subset of registers and store the result in the first free register which is equivalent to the register that was allocated to them. The same principal applies for unary operators but this case is much easier as we can just allocate the first free register to the return of the expression and say that all sub-expressions are allowed to use all free register from above as long as the result is stored in the first free register. We realize that at some point we may run out of registers but this is actually a non-issue as we can just wrap around. Say we have a series of binary operators so that we end up using all of the available 11 free registers. Then say we add another binary operator that expects the result of the initial binary operators. We can basically store the initial result in register 4 and then resolve the result of the next binary operator into register 5 and continue translating our code. Thus, while this register allocation is not the most efficiency, it generates correct code with correct register allocation such that we never run into issues. The main reason why our register allocation strategy works is because we make sure to always store the results of assignment even if it may be redundant and inefficient. So we will never accidentally lose a value from one translated line to another. Furthermore, since there are no global variables in MiniC our job is much easier. Throughout the translation we also make sure to not touch register 11. Overall, we use naive register allocation with wrap-around and possibly redundant memory stores to get correct register allocation. To be more efficient we could have considered dependency analysis or graph coloring but this seems burdensome to do in HACS and we had limited time to complete our implementation.

5 Attributes Used

We use almost all of the provided inherited attributes and pass them to all schemes to generate code. They are as follows.

- ft: This attribute is used to map the expected function type for a function name. We do not make use of it while the provided code does.
- vt: This attribute is used to map a variable identifier to it's stored memory location. We use this make sure a variable is declared before it is used and we can access it using the memory information given.
- return: This attribute contains the label to jump to execute return instructions. This is given to use by the start-up code so we just pass it along until we need to use it any point. Then we make sure the result is stored in register 0 and jump to the relevant point to pop stack contents.
- true: This attribute contains the label to follow when a condition is true. We use this for conditionals statement if-then or while and label the blocks on the statement level. Then the relevant labels are passed to other scheme and these schemes generate code and make sure to jump to the correct labels. For example, `ECond` jumps to the contents of true if a conditional is true. We also rely on HACS to generate unique labels which it accordingly does.
- false: This attribute contains the label to follow when a condition is false. The description is similar to as above.
- value: We do not make use this attribute as the unused attribute serves the same purpose as value in our translation.
- offset: This attribute is used when declaring variables. Variables are stored on the stack with some relative offset to the frame pointer. We assume, as stated before, that the stack grows up so we increment the offset each time a new variable is declared and we always allocate by 4. So we end up allocating some extra space when we declare a `char` variable but this is ok as long the variable is used type-safely. Even though it is implemented inefficiently offset allows us to correctly allocate local variables on the stack.
- unused: This attribute contains the list of unused general purpose registers R4 to R11. Each time we allocate a register for a specific task we make sure to update how this attribute is passed onto the translation of sub-expressions and other statements. Whenever we run out of registers, so when we see match on `NoRs`, we just wrap around allowing us to get correct register allocation.
- next: This attribute is used to indicate what occurs after a given statement is completed. For example, after a while loop is completed if we execute more statements then next is the label of these statement so that when we are done with the while loop we can jump to next and continue with our program. We keep updating and passing this around in the statement translation level.

These are some attributes we added:

- vt2: This is an additional attribute used to pass type information to determine if an identifier is a pointer (or not). If an identifier is a pointer then we can get pointer arithmetic to work but checking for whether a variable is in vt2. Note that all variables in vt2 are in vt but it is not the other way around. So vt2 only contains variables that are declared as pointers.
- num: This is a dummy attribute that allows us to generate unieq labels for `DCI` and `DCS`. This way we can correctly load constants stored into memory and use them correctly. We also get unique labelling which is how we ensure correctness.

6 Work log

Date	Details	Description
April 19	Met up to discuss the assignment. Took an intiial look at things.	Preliminary Meeting
April 26	Decided on our translation plan. Split the workload. Goktug works on statements including IT. Goktug works on assignment expressions. Goktug works on expressions and cond expressions. Kevin works on return expressions. Kevin works on function calls. Kevin works on single statements. Agreed to meet in a week with our results. Agreed to use Git and Overleaf	Translation Logic Workload Assigned
May 3	Met up to discuss our implementation and test. Debugged issues together. Considered how to check function calls but no solution found. Delegated bug fixing tasks. Decided on DCI and DCS issue. Created new attributes vt2 and num. Goktug will implement these attributes. Kevin will fix bugs.	Debugging Solving Issues
May 10	Finalized our implementation. Final debugging and testing. Agreed on our assumptions and tradeoffs being made. Tried more type checking tasks but to no avail. Agreed that our compiler is correct as long as there is a type checker. Delegated tasks for the write-up of the document. Goktug writes translation logic, assumptions and register allocation. Kevin writes attributes used and tests including explanations.	Finalizing the program
May 13	Met up to discuss our implementation one last time. Finished documentation. Wrote introduction. Proof-read one last time. Kevin finished testing.	Just before submitting

7 Testing

We tested our compiler on some of the sample programs that were provided on the first project, in addition to new programs that were created for this project. They are provided in the `samples` and `custom_samples` folders respectively. In this section we go through some of them.

7.1 Sample tests

```
> ./Pr3GoktugKevin.run --scheme=Compile samples/strings.MC
```

```
main   STMFD SP! , {R4-R11, LR}
   MOV R12, SP
   STMFD SP! , {R0-R3}

 dc   DCS  "This string has a \" and \n and \
then it continues with \x68ex and \t\ttabs"

  MOV R4, # 0
   LDR R4, [R4, &dc]
```

```
10    MOV R0, R4
11    B puts
12    MOV R4, R0
13    LDMFD SP! , {R0-R3}
14    STR R4, [R12, -# 4 ]
15
16  dc_69   DCI   0
17    MOV R4, # 0
18    LDR R4, [R4, &dc_69]
19    MOV R0, R4
20    B L
21
22  L   MOV SP, R12
23    LDMFD SP! , {R4-R11, PC}
```

On this program the instruction DCS on line 5 stores a string as consecutive bytes. As mentioned on our assumptions, we do not check if the function puts exists, therefore this program compiles correctly.

```
> ./Pr3GoktugKevin.run --scheme=Compile samples/strlen.MC
```

```
strlen   STMFD SP! , {R4-R11, LR}
  MOV R12, SP

dc   DCI   0
  MOV R4, # 0
  LDR R4, [R4, &dc]
  STR R4, [R12, -# 4 ]

repeat   MOV R4, R0
  MOV R4, R4
  CMP R4, # 0
  BNE body
  B after

body   LDR R4, [R12, -# 4 ]

dc_30   DCI   1
  MOV R5, # 0
  LDR R5, [R5, &dc_30]
  ADD R4, R4, R5
  STR R4, [R12, -# 4 ]
  MOV R4, # 1

dc_44   DCI   1
  MOV R5, # 0
  LDR R5, [R5, &dc_44]
  MUL R4, R4, R5
  MOV R0, R4

after   LDR R4, [R12, -# 4 ]
  MOV R0, R4
  B L

L   MOV SP, R12
  LDMFD SP! , {R4-R11, PC}

```

```
37  main   STMFD SP! , {R4-R11, LR}
38    MOV R12, SP
39    STMFD SP! , {R0-R3}
40
41  dc_92   DCS  "The length of the string is "
42    MOV R4, # 0
43    LDR R4, [R4, &dc_92]
44    MOV R0, R4
45    B puts
46    MOV R4, R0
47    LDMFD SP! , {R0-R3}
48    STR R4, [R12, -# 4 ]
49    STMFD SP! , {R0-R3}
50    STMFD SP! , {R0-R3}
51    MOV R4, R0
52    MOV R0, R4
53    B strlen
54    MOV R4, R0
55    LDMFD SP! , {R0-R3}
56    MOV R0, R4
57    B puti
58    MOV R4, R0
59    LDMFD SP! , {R0-R3}
60    STR R4, [R12, -# 4 ]
61
62  dc_94   DCI   0
63    MOV R4, # 0
64    LDR R4, [R4, &dc_94]
65    MOV R0, R4
66    B L_35
67
68  L_35   MOV SP, R12
69    LDMFD SP! , {R4-R11, PC}
70
```

This is a peculiar program as it demonstrates what happens when you call a function inside a function, in this case `puti(strlen(input))`. On line 53 the function `strlen` is called first and then on line 57 the function `puti` is called.

```
> ./Pr3GoktugKevin.run --scheme=Compile custom_samples/nonlvalue.badMC
```

```
main   STMFD SP! , {R4-R11, LR}
   MOV R12, SP
   '$Print2-Pr3GoktugKevin$Instructions'[
 'Pr3GoktugKevin$Instructions_{_Instructions_}_Instructions_'[
    ...
    ...
    ...
  Pr3GoktugKevin$Instructions_],
 0]
```

The program does not compile because of the assignment expression `2+2 = atoi(in);`. The left side is not an lvalue. Even though this is the job of the typechecker, this is handled by our compiler.

7.2 Additional tests

```
> ./Pr3GoktugKevin.run --scheme=Compile custom_samples/pointer_arithmetic.MC
```

```
1  incrementInt   STMFD SP! , {R4-R11, LR}
2    MOV R12, SP
3    MOV R4, R0
4
5  dc   DCI   1
6    MOV R5, # 0
7    LDR R5, [R5, &dc]
8    ADD R4, R4, R5
9    MOV R0, R4
10   MOV R4, R0
11   MOV R0, R4
12   B L
13
14 L   MOV SP, R12
15   LDMFD SP! , {R4-R11, PC}
16
17 incrementIntPointer   STMFD SP! , {R4-R11, LR}
18   MOV R12, SP
19   MOV R4, # 4
20
21 dc_43   DCI   1
22   MOV R5, # 0
23   LDR R5, [R5, &dc_43]
24   MUL R4, R4, R5
25   MOV R0, R4
26   MOV R4, R0
27   MOV R0, R4
28   B L_7
29
30 L_7   MOV SP, R12
31   LDMFD SP! , {R4-R11, PC}
32
33 incrementCharPointer   STMFD SP! , {R4-R11, LR}
34   MOV R12, SP
35   MOV R4, # 1
36
37 dc_28   DCI   1
38   MOV R5, # 0
39   LDR R5, [R5, &dc_28]
40   MUL R4, R4, R5
41   MOV R0, R4
42   MOV R4, R0
43   MOV R0, R4
44   B L_55
45
46 L_55   MOV SP, R12
47   LDMFD SP! , {R4-R11, PC}
```

The program above shows what happens when you try to increment an integer and when you try to increment a pointer. In the former, the `ADD` instruction is used (on line 8) to increment the integer, while on the latter the `MUL` instruction is used (on line 24 and 40) to multiply the integer with the size of the type that is pointed to.

```
> ./Pr3GoktugKevin.run --scheme=Compile custom_samples/juice.MC
```

```
triple   STMFD SP! , {R4-R11, LR}
  MOV R12, SP
  MOV R4, R0

dc   DCI   3
  MOV R5, # 0
  LDR R5, [R5, &dc]
  MUL R4, R4, R5
  MOV R0, R4
  B L

L   MOV SP, R12
  LDMFD SP! , {R4-R11, PC}

juice   STMFD SP! , {R4-R11, LR}
  MOV R12, SP

dc_65   DCI   0
  MOV R4, # 0
  LDR R4, [R4, &dc_65]
  STR R4, [R12, -# 4 ]

repeat   MOV R4, R0

dc_27   DCI   0
  MOV R5, # 0
  LDR R5, [R5, &dc_27]
  CMP R4, R5
  BGT T
  MOV R4, # 0
  B R

T   MOV R4, # 1

R   CMP R4, R5
  BGT body
  B after

body   LDR R4, [R12, -# 4 ]
  STMFD SP! , {R0-R3}
  LDR R5, [R12, -# 4 ]
  MOV R0, R5
  B triple
  MOV R5, R0
  LDMFD SP! , {R0-R3}
  ADD R4, R4, R5
  STR R4, [R12, -# 4 ]
  MOV R4, R0

dc_69   DCI   1
  MOV R5, # 0
  LDR R5, [R5, &dc_69]
  SUB R4, R4, R5
  MOV R0, R4

after   LDR R4, [R12, -# 4 ]
  MOV R0, R4
```

```
58      B L_14
59
60    L_14   MOV SP, R12
61      LDMFD SP! , {R4-R11, PC}
```

The program above contains two different functions. The first function `triple` multiplies a number by three, which is represented by the instruction `MUL` on line 8. The second function `juice` shows what happens when a while loop is executed and an external function is called. The condition inside the while loop is evaluated on the instruction `CMP` on line 28. If `R4` is bigger than `R5` then it jumps to the block with label `T`.

```
> ./Pr3GoktugKevin.run --scheme=Compile custom_samples/register.MC
```

```
[numbers=left,xleftmargin=5mm]
 firstSeven   STMFD SP! , {R4-R11, LR}
   MOV R12, SP
   MOV R4, R0

 dc   DCI   1
   MOV R11, # 0
   LDR R11, [R11, &dc]
   CMP R4, R11
   BEQ T
   MOV R4, # 0
   B R

 T   MOV R4, # 1

 R   MOV R10, R0

 dc   DCI   2
   MOV R11, # 0
   LDR R11, [R11, &dc]
   CMP R10, R11
   BEQ T_92
   MOV R10, # 0
   B R_53

 T_92   MOV R10, # 1

 R_53   ORR R4, R4, R10
   MOV R9, R0

 dc   DCI   3
   MOV R10, # 0
   LDR R10, [R10, &dc]
   CMP R9, R10
   BEQ T_74
   MOV R9, # 0
   B R_73

 T_74   MOV R9, # 1

 R_73   ORR R4, R4, R9
   MOV R8, R0

 dc   DCI   4
```

```
  MOV R9, # 0
  LDR R9, [R9, &dc]
  CMP R8, R9
  BEQ T_76
  MOV R8, # 0
  B R_49

T_76   MOV R8, # 1

R_49   ORR R4, R4, R8
  MOV R7, R0

dc   DCI   5
  MOV R8, # 0
  LDR R8, [R8, &dc]
  CMP R7, R8
  BEQ T_56
  MOV R7, # 0
  B R_88

T_56   MOV R7, # 1

R_88   ORR R4, R4, R7
  MOV R6, R0

dc   DCI   6
  MOV R7, # 0
  LDR R7, [R7, &dc]
  CMP R6, R7
  BEQ T_91
  MOV R6, # 0
  B R_19

T_91   MOV R6, # 1

R_19   ORR R4, R4, R6
  MOV R5, R0

dc   DCI   7
  MOV R6, # 0
  LDR R6, [R6, &dc]
  CMP R5, R6
  BEQ T_9
  MOV R5, # 0
  B R_86

T_9   MOV R5, # 1

R_86   ORR R4, R4, R5
  CMP R4, # 0
  BNE true
  B false

true
dc_14   DCI   1
```

```
   MOV R4, # 0
   LDR R4, [R4, &dc_14]
   MOV R0, R4
   B L

 false   B after

 after
 dc_18   DCI   0
   MOV R4, # 0
   LDR R4, [R4, &dc_18]
   MOV R0, R4
   B L

 L   MOV SP, R12
   LDMFD SP! , {R4-R11, PC}
```

On this program we test how register allocation is handled. For the statement:

```
   if (x == 1 || x == 2 || x == 3 || x == 4 || x == 5 || x == 6 || x == 7)
```

All registers from `R4` to `R11` are exhausted and not re-utilized. If there was another comparison, the compiler would still manage to properly allocate the registers as outlined in Section 4. Thus, we can have arbitrarily large expressions and as long as we evaluate smaller sub-expressions in some order first then our program will work just fine.