Christopher Pac
N-13242624
cpp270@nyu.edu

Compiler Construction
Spring 2015
Project Milestone 3

TABLE OF CONTENTS

1 Introduction

This document will cover the documentation requirement specified in the Project Milestone 3. It will explain the differences between the provided solutions and implementation, encountered issues, general problems and how they were addressed, lessons learned, and sample tests that were run.

The rest of this section discusses the approach taken while working on this project, the environment used to implement it, and any collaboration that has taken place while working on the “boxes” and code parts of this project.

1.1 General Approach to the Project

I have taken two step approach while working on the code section of this project. In the first step I have implemented my own “boxes” template and in the second step I have simplified it by following the examples in the provided solutions.

My “boxes” template relied on saving intermediate results from the first expression to the stack and retrieving that result off the stack after the second expression. The value from the stack plus the value from the second expression were then combined to calculate the final result. This approach had the benefit of never running out of registers and there was no need to keep track of registers between expressions. That is, each expression could use whatever registers it deemed necessary.

The results of this approach had two immediate negative consequences. It made the generated ARM32 assembly code verbose and difficult to follow. And it also made the assembly code unnecessarily slow by going to the memory for every calculation regardless if there were free registers or not.

The provided solution does not use the stack for intermediate results and just uses available registers. The limitation of this approach is that it is possible to run out of registers on right recursive expressions as illustrated in section 5.1.11.

However, the difficulty to readily check the correctness of the generated ARM32 code and the unlikely case of running out of registers have lead me to adopt the approach of only using registers for simple expressions and not the stack.

1.2 Environment

- Computing environment
  - Linux energon1.cims.nyu.edu

- HACS version
  - HACS 1.0.19

1.3 Collaboration

I did minimal work with Martha Poole. We exchanged some information about the `var` statement while working on the “boxes” part of this project and some simple question/answers about ARM32. However, there was also no significant collaboration on the second part of this project beyond few simple e-mails.

2 Implementation And Issues

This section will cover all the major differences between my implementation and the reference “boxes” solution. I will then briefly describe the layout of the source code and finish by analyzing the issues that I have encountered.

I will not describe every detail of my code and all the limitations as they are well covered in the reference “boxes” solution and my own heavily commented “boxes” solution. For example, the limitation of only being able to pass four arguments is described in both the solution and in the project description. Furthermore, the details of each JST construct, such as activation records, use of offsets, etc, are described very well in the “boxes” solution and it would be redundant to repeat this information here.

Therefore, I will only cover the main differences that I have made while following the provided reference guid.

2.1 Major Reference ARM32 Implementation Differences

The ARM32 code necessary to implement the JST constructs is not complicated and most of the difficulties came from implementing these constructs in HACS.

The differences between the reference and the implementation are mostly due to trying to keep the HACS code simple.

One major difference in my implementation is that the function arguments are saved and kept on the stack. This removes the need to differentiate between variables in registers and variables located on the stack. Since there is no analysis done on the use of function arguments, it is just as valid to keep them on the stack as it is to keep local variables on the stack. This change makes each named variable behave exactly the same. The only difference is at the initialization stage where each function argument is saved to the stack, through the STMFD instruction, and each local variable just increments the SP register.

Another minor difference is in the ARM32 code for the JST `while` construct. The reference solution performs a forward jump to the conditional code, whereas my implementation performs a backwards jump to the conditional code.

The final difference is in the integer expression “box”. The reference solution uses a label to jump over the DCI directive while I move the program counter register into itself, `MOV PC, PC`. This move instruction effectively jumps over the next instruction since PC contains the address of the current instruction plus 8.

2.2 Code Layout

The first part of the code handles pass one where the class and function maps are initialized and the class member offsets are calculated. This code was provided as the starting base for the project.

The second part handles pass two over the JST source code. But, before that an additional map is created to hold the type information and stack offset of each local function variable. All three maps are bundled together in the `Env` data structure. Furthermore, for each map a lookup function was created.

This is followed by the main section of the code where each JST construct recursively outputs the appropriate ARM32 assembly code.

The AsE, TitE, and SeE functions are used to extend the environment for new local or function parameter variables. The AsC, TitC, and SeC functions calculate the new offset from the frame pointer. Unlike the reference solution the SeC subtracts minus four from the current offset when it's the `var` statement as opposed to minus four being passed into it.

The EeT functions determine the type of all the required expression productions. The Identifier and Expression.ID are the only productions of any consequence.

Lastly, the one synthesized attribute is used to obtain a function identifier/name.

2.3 Encountered Issues

I've encountered only few issues while working on this part of the project. The most notable of those was my attempt at creating labels and the use of the Computed sort. All in all, this project implementation was relatively smooth compared to project milestone 2.

2.3.1

The first time I required a label I've attempted to use the Computed sort to concatenate two values together in order to create a unique label. I knew that labels, being symbols, would have the appended randomly generated number. However, I did not know that the system (HACS) would ensure that no clashes would occur between new symbols. Given that, it makes sense that HACS would not allow the manual creation of symbols since these manually created symbols could be duplicates of ones already in existence. I was also not aware, till I've completed most of my project, that fresh symbols could be created on the right side of the function without being present on the left side of a function.

```
Emit(⟦ * ⟨L#⟩ ⟧) → ⟦ s ⟨L Emit(#)⟩ ⟧ ;
```

The `s` variable above is not present on the left side but causes the function to output a fresh label. This information could have potentially simplified my code by avoiding the unnecessary passing of labels to functions and synthesis of labels.

Update
I have simplified the code by creating fresh labels without the synthesizing them or passing them through as function parameters.

2.3.2

The second problem of note was caused by the use of the Computed sort. Initially, I did not have any issues with the Computed sort and the only place where the problem finally occurred was in computing a token concatenation and returning a token back. This was necessary for the `Expression.ID` production in order to lookup the type of a class variable. In order to solve this problem an auxiliary function was required that would force the computed value into a data token value. Once an example of how this is done was provided I was able to solve my problem.

3 Adopted Conventions

This section will briefly reiterate the conventions that were adopted to my own implementation. I have already stated some of these conventions and the reasons for them in section 2.1. This project did not require that a specific convention be strictly followed, as long as a correct ARM32 code was produced. Therefore, not all ARM conventions were followed.

3.1 Empty Object Literal

There are two choices that can be made for an empty object literal list.
1. Don't allocate space since the user did not specify any values and just set the address to null/0.
2. Allocate the required space for this type since each class member item could be set individually using the Exp.id production.

I have adopted the first option since I am working under the assumption that object literals must have all the class members present in order to be valid. Therefore, an empty OL does not have any members and has zero size. Option 2 would be trivial to implement since it's a degenerate case of the OL with values.

3.2 MOV PC, PC

In order to skip the next instruction I used `MOV PC, PC`. Since, PC has the current instruction plus 8, it makes sense that copying it to itself would jump to current instruction plus 8, thus skipping the next instruction. This is used when loading integers greater than 255.

4 Lessons Learned

In section will described some of the lessons learned while implementing my own "boxes" solution and it will cover the bugs discovered in my original solution and how they were addressed. This section will also briefly mention what solutions where taken from the provided reference "boxes" and why.

4.1 Assignment Expression

My initial assignment expression worked under the incorrect assumption that l-val part of the expression would alway return an address and that address would be used as the base address for storing a r-val. However, my l-val expression only returned what was stored on the stack for that variable. This is fine when the value stored is itself an address and it's used with conjunction with an offset to store a value at some particular place in memory, which is not the stack. However, when the variable is a value that needs to be overwritten then my assignment expression would not work correctly.

The solution was to split the assignment expression into two parts that would handle the case when l-val was used as a base address and when it was used as a value to be updated. When it is a value to be updated the base address is the frame pointer and not the l-val.

4.2 `allocate`

It was obvious that some kind of a call was necessary to allocate space on the heap for the object literal. However, it was not initially clear to me that we could simply add this call in the generated ARM assembly code. Once that misunderstanding was cleared up, implementing the said call was straightforward.

4.3 Registers

I have discussed my reasons for changing my initial "boxes" solution in order to avoid saving each intermediate result on the stack in section 1.1. The consequence of that change was the need to more carefully keep track of register usage. Before the change each expression could use whatever registers it deemed necessary but after the change the each successive expression could only use register(s) that was/were not used to save the results of a previous expression.

Furthermore, the R0-R3 became unavailable for general usage.

4.4 Function call

Initially the function call ARM32 code did not take into account that one of the arguments could be a function call itself. Instead, the code used R0-R3 as a destination for each expression in the function argument list. This would have been fine under the old scheme where each result was saved on the stack but it was incorrect under the new scheme. The inner function call will overwrite any previous arguments for the outer function. Therefore, R4 and greater registers can only be used as the target for results from the argument expressions and then those registers need to be moved into R0-R3 just before invoking a function.

5 Testing

All tests were done in the environment specified in section 1.2 Environment. I have tested my solution against custom test and the samples provided in project milestone 3. What follows are

ARM32 code snippets for most significant JST constructs. Most of them are kept as simple as possible to illustrate that correct code is being generated. Where appropriate a brief explanation and comments will be provided.

5.1 Custom Testing

5.1.1 Variable shadowing and Frame Pointer offset for inner scope

```
function void main() {
  var int v1;
  var int v2;
  {
    var int v2;
    v2 = 5;
  }
  v2 = 6;
}
```

```
 main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     ADD SP, R12, # -4
     ADD SP, R12, # -8           // outer scope v2
     ADD SP, R12, # -12          // inner scope v2
     MOV R4, # 5
     STR R4,  [ R12,  # -12  ]   // inner scope offset used
     ADD SP, R12, # -8           // reset SP to location before inner scope
     MOV R4, # 6
     STR R4,  [ R12,  # -8  ]    // outer scope offset used
    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

First v2 is at FP - 8 and second v2 is at FP - 12. When v2 is set to 5 in the inner scope the -12 offset should and is used.
After the inner scope the SP is reset back to what it was before, namely to the the offset of the first v2.
After the inner scope the first v2 is set to 6 and thus -8 should and is used as an offset.

5.1.2 Variable shadowing and Frame Pointer offset with function args

```
function void main(int a1, int a2) {
 var int a1; a1 = 5;
}
```

```
main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     STMFD SP! , {R0}            // a1
     STMFD SP! , {R1}
     ADD SP, R12, # -12          // local var a1
     MOV R4, # 5
     STR R4,  [ R12,  # -12  ]   // correct offset to ‘var’ not to arg
    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

5.1.3 Simple `if` statement

```
function void main() {
  if ( 5 < 6)
   8 ;
}
```

```
main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     MOV R4, # 5
     MOV R5, # 6
     CMP R4, R5
     BLT lt
     B lf
    lt     MOV R4, # 8
    lf    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

If it’s true we jump to label `lt` which just loads 8. If it’s false we jump to label `lf` which is just at the end of the function thus we skip over the load 8 statement.

5.1.4 Simple `if then else` statement

```
function void main() {
  if ( 5 < 6)
   8 ;
  else
   9;
}
```

```
main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     MOV R4, # 5
     MOV R5, # 6
     CMP R4, R5
     BLT lt
     B lf
    lt     MOV R4, # 8
     B ln
    lf     MOV R4, # 9
    ln    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

If its true we jump to label `lt` at load 8 statement and then jump over load 9 statement to the end using label `ln`. If false we jump to label `lf` at stement load 9 and then fall through to the end.

5.1.5 Nested `if then else` statement

```
function void main() {
  if ( 5 < 6)
   if ( 20 > 19 )
```

```
      8 ;
    else
      11 ;
  else
    if ( 50 == 50)
      9;
}
```

```
main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     MOV R4, # 5
     MOV R5, # 6
     CMP R4, R5
     BLT lt
     B lf
    lt     MOV R4, # 20    // start of if ( 20 > 19 )
     MOV R5, # 19
     CMP R4, R5
     BGT lt_59
     B lf_48
    lt_59     MOV R4, # 8  // 8 ;
     B ln
    lf_48     MOV R4, # 11
    ln     B ln_62         // correctly jumps out of the if's
    lf     MOV R4, # 50    // start of if ( 50 == 50)
     MOV R5, # 50
     CMP R4, R5
     BEQ lt_3
     B lf_96
    lt_3     MOV R4, # 9
    lf_96    ln_62     fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

5.1.6 `while` statement

Unlike the reference, this while statement jumps backwards. Therefore, we do the test first and then jump. If the test was true we jump inside the while but if the test was false we jump over the while.

```
function void main() {
  while (5 <= 7)
   8 ;
}
```

```
main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
    b     MOV R4, # 5      // b is the beginning label
     MOV R5, # 7           // do the test
     CMP R4, R5
     BLE lt                  // jump inside if true
     B lf                    // jump out of the while if false
    lt     MOV R4, # 8
     B b                     // jump back to the beginning b
    lf    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

5.1.7 load and store

```
function void main(int v1) {
  var int b1;
  var int b2;
  b2 = v1;
}
```

```
 main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     STMFD SP! , {R0}
     ADD SP, R12, # -8
     ADD SP, R12, # -12
     LDR R4,  [ R12,  # -4  ]       // load v1 at FR - 4 into R4
     STR R4,  [ R12,  # -12  ]      // store R4 into b2 at FR - 12
    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

5.1.8 Object literal initialization and nested class members

```
class Last2 {
  int last;
  int lastlast;
}
class Foo {
  int n;
  Last2 v;           // nested class
  int d;
}

function int main() {
  var Foo f;
  f.v.last = 5;      // setting an inner class member value
}

function int objectLit() {
  var Last2 ls;
  ls = {last:5, lastlast:8};     // object literal init
}
```

```
DEF Last2_last =  0
    DEF Last2_lastlast =  4
    DEF Last2__size =  8
    DEF Foo_n =  0
    DEF Foo_v =  4
    DEF Foo_d =  8
    DEF Foo__size =  12
    main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     ADD SP, R12, # -4              // var Foo f; stores address
     MOV R4, # 5
     LDR R5,  [ R12,  # -4  ]       // 1st load the address of Foo
     LDR R5,  [ R5,  &Foo_v ]       // get the offset address of v = Last2
     STR R4,  [ R5,  &Last2_last ]  // store at Last2 + last offset
    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}

    objectLit     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     ADD SP, R12, # -4              // var Last2 ls; at -4 offset
     STMFD SP! , {R0-R3, R12}
     MOV R0, &Last2__size
     BL allocate
     MOV R4, R0
     LDMFD SP! , {R0-R3, R12}
     MOV R5, # 5
     STR R5,  [ R4,  &Last2_last ]  // R4 has the base address from alloc
     MOV R5, # 8
     STR R5,  [ R4,  &Last2_lastlast ]
     STR R4,  [ R12,  # -4  ]       // save the base address at -4 offset
    fend_99     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

5.1.9 Triple nested class members

```
class Last2 {
  int last;
  int lastlast;
}
class Foo {
  int n;
  Last2 v;
  int d;
}
class Bar {
  int in;
  Foo inner;
}
function int main() {
  var Bar b;
  b.inner.v.last = 5;
}
```

```
DEF Last2_last =  0
    DEF Last2_lastlast =  4
    DEF Last2__size =  8
    DEF Foo_n =  0
    DEF Foo_v =  4
    DEF Foo_d =  8
    DEF Foo__size =  12
    DEF Bar_in =  0
    DEF Bar_inner =  4
    DEF Bar__size =  8
    main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     ADD SP, R12, # -4
     MOV R4, # 5
     LDR R5,  [ R12,  # -4  ]       // base address
     LDR R5,  [ R5,  &Bar_inner ]   // offset to inner which is Foo class
     LDR R5,  [ R5,  &Foo_v ]       // offset to v which is Last2 class
     STR R4,  [ R5,  &Last2_last ]  // offset to last which is an int
    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

5.1.10 Empty Object Literal

See section 3.1 for more information.

```
class Foo {
}

function void main(Foo f) {
  f = {};
}
```

```
DEF Foo__size =  0
    main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     STMFD SP! , {R0}
     MOV R4, # 0            // nothing to allocate since empty
     STR R4,  [ R12,  # -4  ]
    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

5.1.11 `return` Computed expression and Register allocation limitation

The code below right recurses and illustrates the limitation of register allocation implementation. In this example it is clear how we could run out of registers if the nested parentheses calculations continued.

```
function int main() {
  return (7+(8*(9-(11*12))));
}
```

```
main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     MOV R4, # 7
     MOV R5, # 8
     MOV R6, # 9
     MOV R7, # 11
     MOV R8, # 12
     MUL R7, R7, R8         // 11 * 12
     SUB R6, R6, R7         // 9 - result
     MUL R5, R5, R6         // 8 * result
     ADD R4, R4, R5           // 7 + result
     MOV R0, R4               // move result to R0
     B fend                   // return
    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

5.1.12 Short-circuiting behavior and AND OR operators

```
function void main() {
  if ( 6 > 7 && 8 <=9 )
   return;

  if ( 20 >= 19 || 20 != 21)
   return;

  var int v;

  if ( 20 <= 19 || 20 != 21 && v >= 200)
    return;
}
```

```
main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     MOV R4, # 6
     MOV R5, # 7
     CMP R4, R5
     BGT lnt
     B lf
    lnt     MOV R4, # 8          // jump 2nd part of AND if 1st true
     MOV R5, # 9
     CMP R4, R5
     BLE lt
     B lf
    lt     B fend                // return
    lf     MOV R4, # 20
     MOV R5, # 19
     CMP R4, R5
     BGE lt_67                   // jump all the way to return if true
     B lnf
    lnf     MOV R4, # 20         // jump to 2nd part of OR if 1st false
     MOV R5, # 21
     CMP R4, R5
     BNE lt_67
     B lf_44
    lt_67     B fend             // return
    lf_44     ADD SP, R12, # -4   // var int v;
     MOV R4, # 20
     MOV R5, # 19
     CMP R4, R5
     BLE lt_45                   // if true jump to return
     B lnf_6                    // if false do the AND
    lnf_6     MOV R4, # 20
     MOV R5, # 21
     CMP R4, R5
     BNE lnt_42
     B lf_48
    lnt_42     LDR R4,  [ R12,  # -4  ]
     MOV R5, # 200
     CMP R4, R5
     BGE lt_45
     B lf_48
    lt_45     B fend              // return
    lf_48    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

5.1.13 Logical NOT operator

```
function void main() {
  if (! (6 == 7))
   return ;
}

main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     MOV R4, # 6
     MOV R5, # 7
     CMP R4, R5
     BEQ lf
     B lt
    lt     B fend
    lf    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

Labels are correctly swapped.

5.1.14 x = fn(a1, v = fn(), a3, a4)

The bug found while testing will be left here as a reference. The bug is caused by a function parameter itself being a function. The fixed ARM32 code is provided after the bug reference example.

```
function int sum(int a1, int a2, int a3, int a4) {
  return a1 + a2 + a3 + a4;
}
function void simple() {
  var int n;
  n  = sum(3,4,5,6);
}

function void nested_fn() {
  var int n;
  var int v;
  n = sum(3, v = sum(1,2,3,4), 4,5,-256); // Too Many Parameters
}
```

Bug in this ARM32 code. See below for fixed version.

```
sum     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     STMFD SP! , {R0}               // a1
     STMFD SP! , {R1}               // a2
     STMFD SP! , {R2}               // a3
     STMFD SP! , {R3}               // a4
     LDR R4,  [ R12,  # -4  ]       // a1 load
     LDR R5,  [ R12,  # -8  ]       // a2 load
     ADD R4, R4, R5
     LDR R5,  [ R12,  # -12  ]      // a3 load
     ADD R4, R4, R5
     LDR R5,  [ R12,  # -16  ]      // a4 load
     ADD R4, R4, R5
     MOV R0, R4
     B fend
    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
simple     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     ADD SP, R12, # -4
     STMFD SP! , {R0, R1, R2, R3, R12}
     MOV R0, # 3                    // param 1
     MOV R1, # 4                    // param 2
     MOV R2, # 5                    // param 3
     MOV R3, # 6                    // param 4
     BL sum                         // call function
     MOV R4, R0
     LDMFD SP! , {R0, R1, R2, R3, R12}
     STR R4,  [ R12,  # -4  ]       // save results to n at FP-4
    fend_12     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
nested_fn     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     ADD SP, R12, # -4
     ADD SP, R12, # -8
     STMFD SP! , {R0, R1, R2, R3, R12}
     MOV R0, # 3                    // param 1 for outer function
     STMFD SP! , {R0, R1, R2, R3, R12}
     MOV R0, # 1                    // param 1 for inner function
     MOV R1, # 2                    // param 2 for inner function
     MOV R2, # 3                    // param 3 for inner function
     MOV R3, # 4                    // param 4 for inner function
     BL sum
     MOV R1, R0             // BUG!!!! R1 for outer will be overwritten
     LDMFD SP! , {R0, R1, R2, R3, R12}
     STR R1,  [ R12,  # -8  ]
     MOV R2, # 4
     MOV R3, # 5
     LDR R4,  [ PC,  # 0  ]
     MOV PC, PC
    DCI   256
     RSB R4, R4, # 0
     BL sum
     MOV R4, R0
     LDMFD SP! , {R0, R1, R2, R3, R12}
     STR R4,  [ R12,  # -4  ]
    fend_18     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

Fixed version of ARM32 with the same JST code. Now each argument expression is not directly stored into R0-R3.

```
sum     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     STMFD SP! , {R0}
     STMFD SP! , {R1}
     STMFD SP! , {R2}
     STMFD SP! , {R3}
     LDR R4,  [ R12,  # -4  ]
     LDR R5,  [ R12,  # -8  ]
     ADD R4, R4, R5
     LDR R5,  [ R12,  # -12  ]
     ADD R4, R4, R5
     LDR R5,  [ R12,  # -16  ]
     ADD R4, R4, R5
     MOV R0, R4
     B fend
    fend     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
simple     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     ADD SP, R12, # -4
     STMFD SP! , {R0, R1, R2, R3, R12}
     MOV R4, # 3
     MOV R5, # 4
     MOV R6, # 5
     MOV R7, # 6
     MOV R0, R4
     MOV R1, R5
     MOV R2, R6
     MOV R3, R7
     BL sum
     MOV R4, R0
     LDMFD SP! , {R0, R1, R2, R3, R12}
     STR R4,  [ R12,  # -4  ]
    fend_77     MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
nested_fn     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
     MOV R12, SP
     ADD SP, R12, # -4
     ADD SP, R12, # -8
     STMFD SP! , {R0, R1, R2, R3, R12}
     MOV R4, # 3
     STMFD SP! , {R0, R1, R2, R3, R12}
     MOV R5, # 1
     MOV R6, # 2
     MOV R7, # 3
     MOV R8, # 4
     MOV R0, R5
     MOV R1, R6
     MOV R2, R7
     MOV R3, R8
     BL sum
     MOV R5, R0
     LDMFD SP! , {R0, R1, R2, R3, R12}
     STR R5,  [ R12,  # -8  ]
     MOV R6, # 4
     MOV R7, # 5
     LDR R8,  [ PC,  # 0  ]
     MOV PC, PC
    DCI   256
     RSB R8, R8, # 0
     MOV R0, R4
     MOV R1, R5
     MOV R2, R6
     MOV R3, R7
     MOV R4, R8       // JST source has too many parameters
     BL sum
     MOV R4, R0
     LDMFD SP! , {R0, R1, R2, R3, R12}
     STR R4,  [ R12,  # -4  ]
    fend_2    MOV SP, R12
     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
```

5.1.14 fn().foo when function returns a class with members

```
class Foo {
  int v;
  int g;
}
function Foo test() {
  var Foo f;
  return f;
}
function void main() {
  var int in;
  in = test().v;
}
```

```
DEF Foo_v =  0
    DEF Foo_g =  4
    DEF Foo__size =  8
    test     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
```

```
 MOV R12, SP
 ADD SP, R12, # -4
 LDR R4,  [ R12,  # -4  ]
 MOV R0, R4
 B fend
fend     MOV SP, R12
 LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}
main     LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, LR}
 MOV R12, SP
 ADD SP, R12, # -4
 STMFD SP! , {R0, R1, R2, R3, R12}
 BL test
 MOV R4, R0               // R4 has the base address of Foo
 LDMFD SP! , {R0, R1, R2, R3, R12}
 LDR R4,  [ R4,  &Foo_v ] // offset from the base address to v loc
 STR R4,  [ R12,  # -4  ]
fend_78     MOV SP, R12
 LDMFD SP! , {R4, R5, R6, R7, R8, R9, R10, R11, PC}  OK
```

5.2 Sample Testing

All three files were tested and appeared to generate the correct ARM32 code.

5.2.1 Fib.jst

Looks ok.

5.2.2 FastFib.jst

There is a JST syntax error at line 17
`var Pair next;`

Pair is an undefined type.

There is also a bug in the ARM32 code. The multiple call to function `allocate` causes HACS to generate a unique label allocate_##. This is discussed in outstanding issues section 6.1.

5.2.3 Div.jst

Looks ok.

6 Outstanding Issues and Possible Solutions

6.1 allocate converted to a symbol

The syntax for `BL` machine instruction specifies that the target is an `Identifier`. However, since an Identifier is a HACS symbol, HACS will create a new unique identifier each time. Therefore, if there are multiple calls to allocate and the code simple does `BL allocate` the allocate is treated as a new symbol and HACS will create allocate_##. The two solutions that would work are:

1. Extend the Identifier sort with allocate: `sort Identifier | symbol ⟦⟨ID⟩⟧ | ⟦ allocate ⟧;`
2. Add allocate as a global identifier to the environment.

I have tested the first solution and it appears to be working as desired. However, I did not include the first solution or the second solution as a fix since the fix would touch every aspect of the program. Therefore, this fix would require extensive retesting and I didn't not feel comfortable making the changes this close to the deadline. Following Prof. Rose's advice I'm reporting this as a "quirk" with the two possible solutions.