

# **Tiger to RISC V Compiler**

*A Project Report Submitted  
in Partial Fulfillment of the Requirements  
for the Degree of*

**Bachelor of Technology**

*by*

**Sourabh Aggarwal**  
(111601025)

*under the guidance of*

**Dr. Piyush P. Kurur**



INDIAN INSTITUTE  
OF TECHNOLOGY  
**PALAKKAD**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

# CERTIFICATE

*This is to certify that the work contained in this thesis entitled “**Tiger to RISC V Compiler**” is a bonafide work of **Sourabh Aggarwal (Roll No. 111601025)**, carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my supervision and that it has not been submitted elsewhere for a degree.*

**Dr. Piyush P. Kurur**

Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

# Acknowledgements

I would like to express my sincere gratitude to my mentor Dr. Piyush P. Kurur for providing his guidance, comments and suggestions throughout this semester.

Also I would like to thank Benjamin Landers for his awesome RISC simulator and also for helping me on various occasions on how to use his tool best suited for my application.

# Contents

<b>List of Figures</b>	<b>1</b>
<b>1 Introduction</b>	<b>1</b>
1.1 What has been achieved this Semester . . . . .	1
1.2 Road Ahead . . . . .	6
1.3 Organization of The Report . . . . .	7
<b>2 Introduction to RISC V and Runtime</b>	<b>9</b>
2.1 Choosing a Simulator . . . . .	9
2.2 Basic Examples . . . . .	10
2.2.1 Hello World . . . . .	10
2.2.2 Saving callee save registers . . . . .	12
2.3 Runtime . . . . .	14
<b>3 Phase I : Constructing Abstract Syntax Tree</b>	<b>15</b>
3.1 Linking Lexer and Parser . . . . .	15
3.2 Intermediate Error Handling . . . . .	16
3.3 Lexer . . . . .	17
3.4 Key Map . . . . .	19
3.5 Symbols in Tiger . . . . .	20
3.6 Parser . . . . .	20

<b>4</b>	<b>Phase II : Constructing Intermediate Representation Tree</b>	<b>23</b>
4.1	Type Interface . . . . .	23
4.2	Types/Variables Environment . . . . .	24
4.3	Finding Escaped Variables . . . . .	24
4.4	Intermediate Tree Representation . . . . .	25
4.5	Understanding Functions Calls in Tiger . . . . .	27
4.6	RISC Frame . . . . .	29
4.7	Translating to IR . . . . .	29
4.8	Semantic Analysis . . . . .	31
<b>5</b>	<b>Phase III : Generating Assembly Code</b>	<b>33</b>
5.1	Canonisation . . . . .	33
5.1.1	Abstract . . . . .	33
5.1.2	Why CALL nodes are an issue? . . . . .	34
5.1.3	Why ESEQ nodes are an issue? . . . . .	34
5.2	Instruction Selection . . . . .	34
5.2.1	Instruction Representation . . . . .	36
5.2.2	Code Generation . . . . .	37
5.2.3	Maximal Munch Algorithm . . . . .	38
5.3	Liveness Analysis and Interference Graph . . . . .	39
5.3.1	Liveness Analysis . . . . .	39
5.3.2	Interference Graph . . . . .	40
5.4	Register Allocation . . . . .	41
5.4.1	Algorithm . . . . .	41
	<b>References</b>	<b>43</b>

# List of Figures

1.1	Lexer detecting error where newline is inserted in a string . . . . .	2
1.2	Website: tigercompiler.ml . . . . .	2
1.3	Automated testing using Travis . . . . .	3
1.4	As many function arguments possible . . . . .	3
1.5	Git commit showing addition of this feature . . . . .	4
1.6	Arithmetic Shift Operations . . . . .	4
1.7	Multiplication Optimization . . . . .	5
5.1	ESEQ Removal. Image source: [1] . . . . .	35
5.2	Tree Patterns showing arithmetic and memory instructions. Image source: [1]	37
5.3	A tree tiled in two ways. Image source: [1] . . . . .	38

# Chapter 1

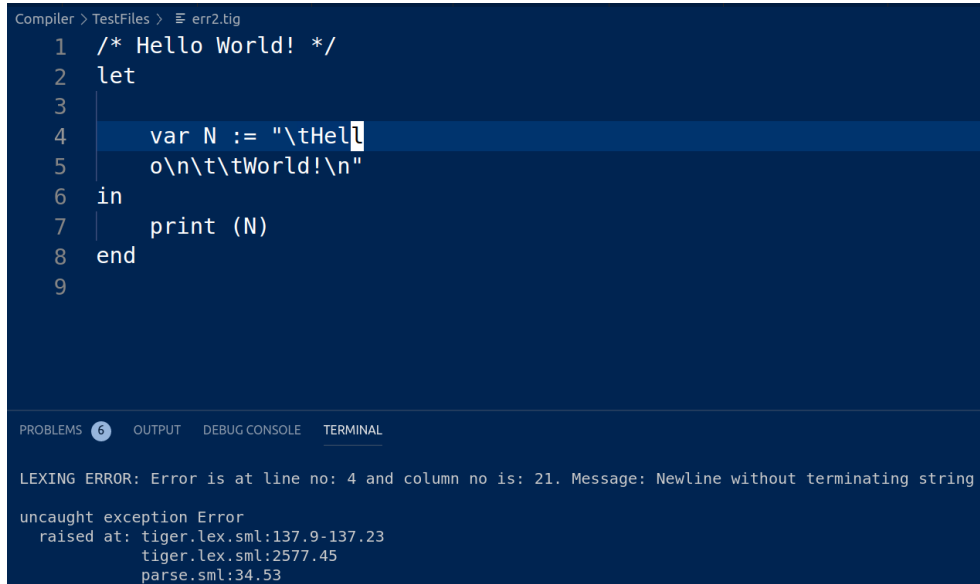
## Introduction

During 6<sup>th</sup> Semester, I wrote a compiler to compile Tiger to MIPS. Now is my attempt to build upon this compiler, to improve its functionality, efficiency and fix various issues/bugs. Also now instead of MIPS, I'll be compiling to RISC V.

### 1.1 What has been achieved this Semester

1. Successfully translated compiler functionality from MIPS to RISC V.
  - (a) Now the compiled code is generated based on RISC V machine and corresponding ISA.
  - (b) During this process, lots of code refactoring is done along with improvement in time complexity of various intermediate computations. Like instead of finding an element in a list, a red black map is used, etc.
  - (c) Files such as runtime.s and riscframe.sml were completely rewritten along with various modifications required at other places.
2. Implemented improvements in lexical phase to detect more errors; errors in lexical phase are reported immediately resulting in program termination unlike in semantic

analysis where a guess is made to facilitate printing all errors in the end. See figure 1.1 for an example.



```
Compiler > TestFiles > err2.tig
1  /* Hello World! */
2  let
3
4  var N := "\tHello
5  o\n\t\tWorld!\n"
6
7  in
8    print (N)
9  end

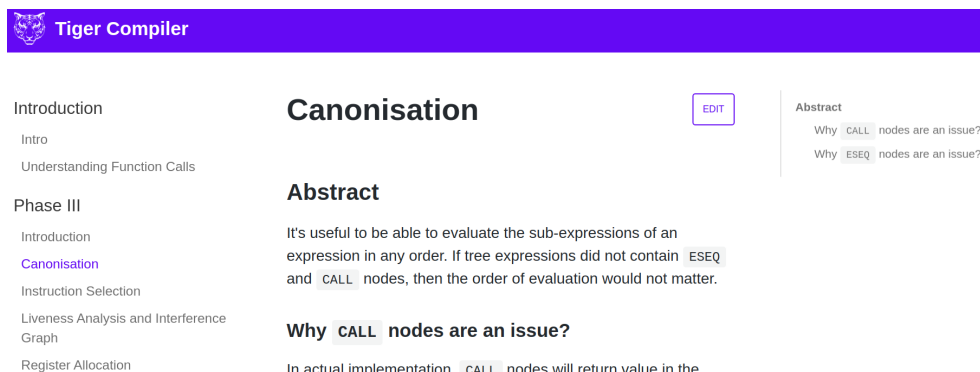
PROBLEMS 6 OUTPUT DEBUG CONSOLE TERMINAL

LEXING ERROR: Error is at line no: 4 and column no is: 21. Message: Newline without terminating string

uncaught exception Error
  raised at: tiger.lex.sml:137.9-137.23
            tiger.lex.sml:2577.45
            parse.sml:34.53
```

**Fig. 1.1:** Lexer detecting error where newline is inserted in a string

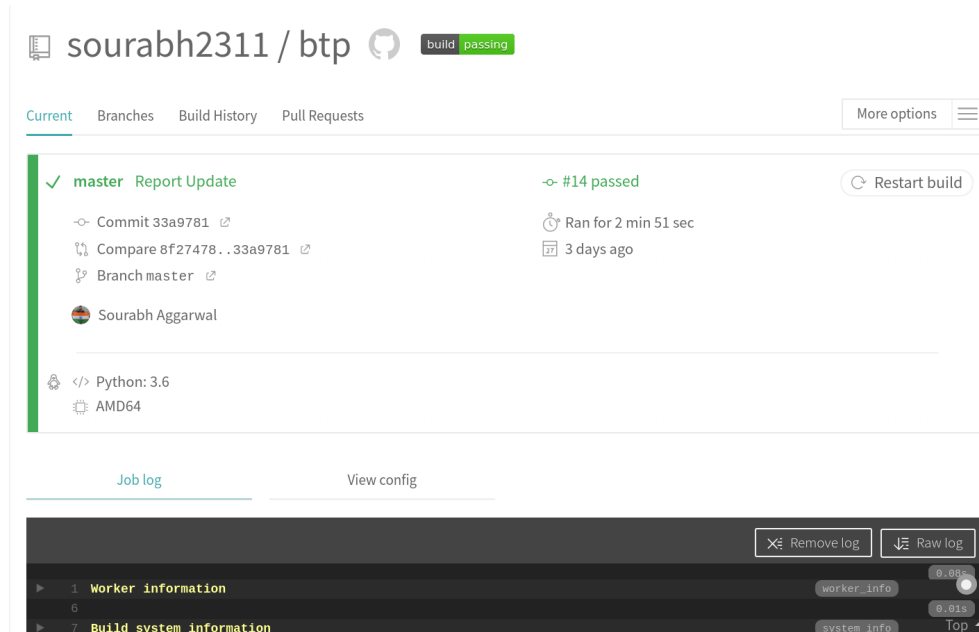
3. Wrote complete documentation of my compiler at tigercompiler.ml. This is done to help me and anyone interested in this project to quickly revise the fundamentals and understand the working of this compiler. Figure 1.2 shows image of the site.



**Fig. 1.2:** Website: tigercompiler.ml

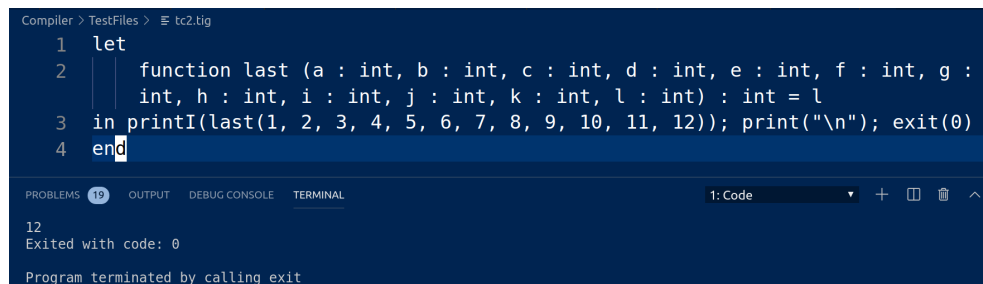
4. Wrote automated testing using Travis. See this. Now I'll be able to see whether my changes don't break the existing functionalities and also it is useful in case someone sends a pull request. Figure 1.3 shows my project at Travis.





**Fig. 1.3:** Automated testing using Travis

5. **Fixed** a major bug; Initially my compiler supported only fixed number of arguments (same as number of argument registers in the machine). Now this has been extended to support any number of arguments (see fig 1.5). During this process I have as well figured out how to completely remove **fp** register as it is as such obsolete. This will be done in future. Figure 1.4 shows an example where a lot many arguments are passed to a function and the last argument is returned.

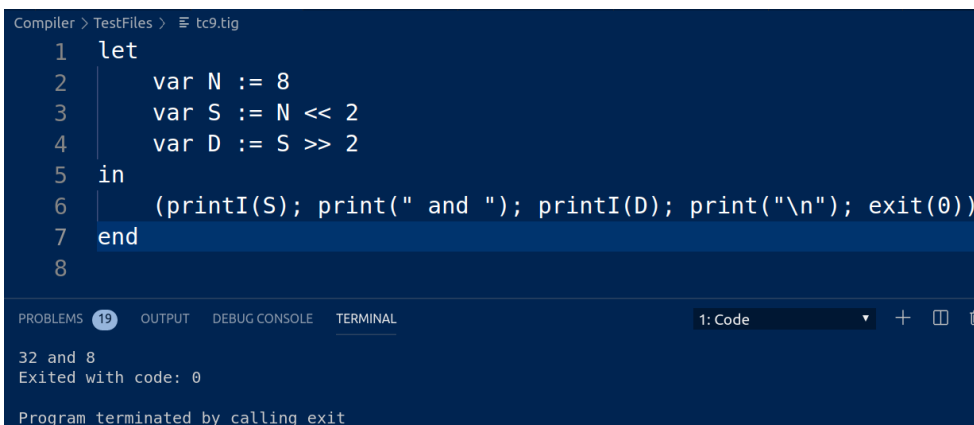


**Fig. 1.4:** As many function arguments possible

6. Added 2 more arithmetic operations, viz. **left shift** and **right shift**. Figure 1.6 shows an example usage of these operations.



**Fig. 1.5:** Git commit showing addition of this feature



**Fig. 1.6:** Arithmetic Shift Operations

```
Compiler > TestFiles > tc10.tig
1 let
2   var N := 8
3   var M := 4 * N
4   var O := N * 4
5 in
6   (printI(M); print("\n"); printI(O); print("\n"); exit(0))
7 end
8

PROBLEMS 19 OUTPUT DEBUG CONSOLE TERMINAL 1: Code
32
32
Exited with code: 0
```

(a) Code

```
Compiler > TestFiles > ASM tc10.tig.s
28 mv s10, s11
29 li s7, 8
30 slli s1, s7, 2
31 mv s11, s1
32 slli s1, s7, 2
33 mv a0, s11
34 jal printI
```

(b) mul statements replaced with sll

**Fig. 1.7:** Multiplication Optimization

7. Implemented multiplication by power of 2 optimization inside basic block thus laid foundation for other basic block optimizations like constant propagation, constant folding. An example of multiplication by power of two optimization is shown in figure 1.7a and 1.7b.

8. Started work on giving a guess of literal in case of small typo.

- Thinking of printing suggestions which are atmost 2 distance apart. This will bring the time complexity of standard DP approach of  $O(n^2)$  to just  $O(n)$ .
- Currently the issue is that to implement this, I would have to do lots of modification of the current code. Although I have abstracted out *Not Found* error messages out with the environment, what is just left is to compare the literal

with those of nearby length in the environment.

- But to efficiently get those strings of nearby length there should be a map which maps lengths to the string set. (An array indexed by length may as well be used) To do this, I would have to define additional data structures and put them at appropriate place.
- The main issue is that in the current design, environment just have integers mapped to environment entry. We got this integer by mapping string to counter, not storing the reverse map. This can be worked around by using Atom.

9. Started work on improving my Register Allocator. Current version is a bit simplified version of the algorithm mentioned in the text and is without coalescing.

## 1.2 Road Ahead

Some among many possible improvements are mentioned below:-

- Current implementation of register allocation isn't completely as what is given in the book. My register allocator lacks coalescing and is therefore incomplete. This algorithm was written in hurry last semester and is to be implemented with better heuristics.
- Basic blocks has to optimized with optimizations like constant propagation, constant folding, strength reduction etc.
- Error messages has to improved in semantic phase. Possible improvement can be to give suggestion for basic typos.
- String comparison has to be made as simple as "str1 > str2", etc., instead of calling the string comparison functions to determine it.
- To implement ability to include pre-written code (header) files.

- To implement garbage collection.
- To implement dataflow analyses such as reaching definitions and available expressions and use them to implement some of the optimizations.
- To implement first-class function values in Tiger, so that functions can be passed as arguments and returned as results.
- Add support for compile time (initial) arguments.
- And much more is possible, its like in a product.

### **1.3 Organization of The Report**

The code written for this compiler is enormous and this report gives documentation on each of these files in an understandable order. But before proceeding it is important to read about Tiger language from this book (read Tiger Reference Manual in Appendix).



# Chapter 2

## Introduction to RISC V and Runtime

### 2.1 Choosing a Simulator

In case of MIPS, it was straight forward to compile and run the assembly code but in case of RISC V, I had to struggle for some time as I couldn't find an appropriate documentation for it. After many iterations, I found the most suitable simulator, namely, RARS.

Please see the corresponding description of system calls [here](#). It as well have a nice companion documentation which I expect one to read before proceeding further with the report.

There are other ways to compile and run RISC V code:

- Venus, Github repo. Note that for system calls, their argument register is different, see [this](#).
- **spike** (sort of an official simulator), installed when using `riscv-gnu-toolchain`. Note it was as well required to install `pk`. System calls are different than RARS, basically it follows linux system calls. Can see these system calls [here](#) and [here](#). And linux system calls [here](#), note that system calls of interest can be concisely seen [here](#).

So to compile and run the program, do: (Don't know if this is the intended way but after a lot of trial and error, I found this)

```
>> riscv64-unknown-elf-as -o filename.o filename.s
>> riscv64-unknown-elf-ld -o filename filename.o
>> spike pk filename
```

## 2.2 Basic Examples

Few things to keep in mind:

- Note that don't use `$reg`, instead simply use `reg`.
- Always have two sections, one for data and another for text.
- During `ecall` all registers besides the output are guaranteed not to change.
- Put return value in `a0`.
- When we want our register value to be saved across system call, we can simply put it in stack instead of using registers  $s_i$  as anyway we would have to save their value first in stack so no change in overhead.
- `a7` is used to tell which system call.
- Save `ra` register if executing `jal` inside a function.

### 2.2.1 Hello World

```
.data # Tell the assembler we are defining data not code

msg: # Label this position in memory so it can be referred to in our code
    .string "hello world" # Copy the string "hello world" into memory

.text # Tell the assembler that we are writing code (text) now
```



```

start:

    li a7, 4 # li means to Load Immediate and we want to load the value 4
              into register a7

    la a0, msg # la is similar to li, but works for loading addresses

    ecall

    li a7, 10 # Exit call

    ecall

```

To get the same code working using spike.

```

.globl _start # We must need to give _start, .globl helps to see it
              outside this file

.data

str:

    .string "Hello World!\n"

.text

_start:

    li a0, 1

    la a1, str

    li a2, 13 # length of the string as required for linux system call. We
              can write a function which will determine the length of the string by
              checking for terminating null character.

    li a7, 64

    ecall

    li a0, 0 # The exit code we will be returning is 0

```

```
li a7, 93  # Again we need to indicate what system call we are making  
and this time we are calling exit(93)  
ecall
```

## 2.2.2 Saving callee save registers

```
.data  
  
bef: .string "Before modification, value is: "  
dur: .string "\nInside function, value is: "  
aft: .string "\nAfter function call, value is: "  
  
.text  
  
main:  
    addi s0, zero, 1  
    # Print bef  
    la a0, bef  
    li a7, 4  
    ecall  
    # Print int  
    li a7, 1  
    mv a0, s0  
    ecall  
    jal increment  
    # Print aft  
    la a0, aft  
    li a7, 4
```

```

ecall
# Print int
li a7, 1
mv a0, s0
ecall
# Exit
li a7, 10
ecall

```

increment:

```

addi sp, sp, -4
sw s0, 0(sp) # '0' denotes the offset, in case of 0, we can simply omit
it.
addi s0, s0, 1
# Print string
la a0, dur
li a7, 4
ecall
# Print the incremented integer
mv a0, s0
li a7, 1
ecall
lw s0, 0(sp)
addi sp, sp, 4
jr ra

```

## 2.3 Runtime

Associated File(s) Link
<code>runtime.s</code>
<i>Please see comments in the file(s) for more details.</i>

There are some standard functions which our Tiger program can use (like `print`, etc.). They are written in `runtime.s`. I could have as well used file provided by author but as I want to augment it further and implement things my way, I decided to write runtime myself. Each of the functions of runtime is explained in the code for easy understanding.

# Chapter 3

## Phase I : Constructing Abstract Syntax Tree

This chapter deals with construction of Abstract Syntax Tree (AST), i.e. Lexical Analysis and Parsing are explained herein.

### 3.1 Linking Lexer and Parser

#### Associated File(s) Link

`parse.sml`

*Please see comments in the file(s) for more details.*

We know the compiler follows "Lexical Analysis  $\rightarrow$  Parsing  $\rightarrow \dots$ ". `parse.sml` is the one which does these two and ties the various files associated with it and finally returns the desired abstract syntax tree.

## 3.2 Intermediate Error Handling

Associated File(s) Link
<a href="#">errormsg.sml</a>
<i>Please see comments in the file(s) for more details.</i>

Besides errors encountered in lexical analysis phase, they'll be printed using `errormsg.sml` which will print these errors in an elaborate manner.

I'll be explaining the signature of this file here, which is sufficient to understand the working of this file.

```
(* In our lexer, each line will occur at a particular "pos" which is
incremented for each character ("pos" is nothing but the number of
characters read till now). For each line, we maintain its "pos". Our goal
is that given "pos", we have to determine the line number and column
number, this can now be easily done, just determine that line with
maximum "pos" which is less than the given "pos", the difference now
gives the column number. Reason of doing this in such an odd way is
because in our grammar, lexer and in abstract syntax file we define "pos"
to be int and not as a tuple (int, int) where first parameter could
denote line number and other one as column number. *)
```

```
signature ERRORMSG =
```

```
sig
```

```
  val anyErrors : bool ref  (* has there been an occurrence of any error?
*)
```

```
  val fileName : string ref (* Updated in parse.sml, used when printing
errors *)
```

```
  val lineNum : int ref (* Number of lines in the read file *)
```

```

val linePos : int list ref (* as defined in the top most comment, it
is the list containing value of "pos" at each line *)

val error : int -> string -> unit (* it should take "pos" and error
message to print, from "pos" it will determine the line number and
the column number, and will print abstractly
"filename:lineNo.ColNo:Message". *)

exception Error

val impossible : string -> 'a    (* raises Error, for a behavior we
didn't expect *)

val reset : unit -> unit    (* reset the parameters, so that we can move
on to read new file *)

end

```

### 3.3 Lexer

#### Associated File(s) Link

tiger.lex

*Please see comments in the file(s) for more details.*

tiger.lex is simply a lexer for our language, just that for some of the errors like non terminated string, etc. are easily detected in this phase and thus are printed now with putting an end to the compilation phase. Usually one detects many errors and print them all instead of just printing one error and stopping. But I feel that errors in lexical phase, if found, are critical and should be handled first most.

#### Guidelines

Few important rules to keep in mind when writing a ML-Lex file:

- Longest match: The longest initial substring of the input that can match any regular

expression is taken as the next token.

- Rule priority: For a particular longest initial substring, the first regular expression that can match determines its token type. This means that the order of writing down the regular-expression rules has significance.

- An individual character stands for itself, except for the reserved characters

? \* + | ( ) ^ / ; . = < > [ { " \ \$

- A backslash followed by one of the reserved characters stands for that character.
- Inside the brackets, only the symbols

\ - ^

are reserved. An initial up-arrow ^ stands for the complement of the characters listed, e.g. [^abc] stands any character except a, b, or c.

- To include ^ literally in a bracketed set, put it anywhere but first; to include - literally in a set, put it first or last.
- The dot . character stands for any character except newline, i.e. the same as

[^\n]

- The following special escape sequences are available, inside or outside of square brackets:

\b backspace

\n newline

\t horizontal tab

\ddd where ddd is a 3 digit decimal escape



- Any regular expression may be enclosed in parentheses ( ) for syntactic (but, as usual, not semantic) effect
- A sequence of characters will stand for itself (reserved characters will be taken literally) if it is enclosed in double quotes " ".
- A postfix repetition range {a, b} where a and b are small integers stands for any number of repetitions between a and b of the preceding expression. The notation {a} stands for exactly a repetitions. Ex: [0-9]{3} Any three-digit decimal number.
- The rules should match all possible input. If some input occurs that does not match any rule, the lexer created by ML-Lex will raise an exception `LexError`.

### 3.4 Key Map

#### Associated File(s) Link

`table.sml`

`table.sig`

*Please see comments in the file(s) for more details.*

For our immediate steps, we would need a way to map our "keys" to a particular "value", for this we create a functor, "IntMapTable" which takes the type of "key" and a function to get an integer corresponding to that "key". Then this functor uses "IntBinaryMap", a hash map to represent our map. This "IntBinaryMap" is polymorphic, so we can use it for any value corresponding to our int (which we obtained from our key). This is the purpose served by `table.sig` and `table.sml`.

### 3.5 Symbols in Tiger

#### Associated File(s) Link

symbol.sml

*Please see comments in the file(s) for more details.*

Each of the variable, function declaration etc. will be stored as "symbols" (a pair of (string, int)). Where each symbol will have a unique integer. We can then map this integer to anything using our "IntBinaryMap". Purpose of keeping this symbols is that we would like to see whether this variable is declared before or not, whether this type exists or not, etc.

### 3.6 Parser

#### Associated File(s) Link

tiger.grm

absyn.sml

*Please see comments in the file(s) for more details.*

`tiger.grm` is the parser for our language, it will read the tokens from lexer and build AST using `absyn.sml`. Grammar directly follows the rules given in the book.

#### Guidelines

- Format of ML-YACC: user declarations %% parser declarations %% grammar rules.
- By default, ML-YACC resolves shift-reduce conflicts by shifting, and reduce-reduce conflicts by using the rule that appears earlier in the grammar. If then (else) shift-reduce conflict is thus not harmful.
- Consider for example:

$E \rightarrow E * E. \quad (+)$

$E \rightarrow E. + E \quad (\text{any})$

The precedence declarations (`%left`, etc.) give priorities to the tokens; the priority of a rule is given by the last token occurring on the right-hand side of that rule. Thus the choice here is between a rule with priority `"*"` and a token with priority `"+"`; the rule has higher priority, so the conflict is resolved in favor of reducing.

- When the rule and token have equal priority, then a `%left` precedence favors reducing, `%right` favors shifting, and `%nonassoc` yields an error action.
- Instead of using the default "rule has precedence of its last token," we can assign a specific precedence to a rule using the `%prec` directive. This is commonly used to solve the "unary minus" problem. In most programming languages a unary minus binds tighter than any binary operator, so `—6 * 8` is parsed as `(—6) * 8`, not `—(6 * 8)`.
- Precedence declaration should be written in the order of increasing precedence.

For more description, please see the comments in the file `tiger.grm`.



# Chapter 4

## Phase II : Constructing Intermediate Representation Tree

Now that we have constructed our Abstract Syntax Tree, we will now have to convert it to an Intermediate Representation to solve  $m \times n$  problem and also code generation would be easy on this language as it is more close to many machine languages.

### 4.1 Type Interface

Associated File(s) Link

`types.sml`

*Please see comments in the file(s) for more details.*

Our next step is to do type checking of our input AST. For this purpose we need to define auxiliary types. Thus, we have `types.sml` representing the same. Thus now we can easily map a "symbol" to its corresponding type.

## 4.2 Types/Variables Environment

### Associated File(s) Link

`env.sml`

`env.sig`

`temp.sml`

`temp.sig`

*Please see comments in the file(s) for more details.*

Now for type checking, we need some environment under which we have to check compatibility of types as for instance, environment can be different for different functions as within `let` block of each function there can be new declarations etc. By default we have some standard types and functions already present in our environment and are thus listed in `env.sml`. Now each of these functions will correspond to some label in our MIPS code, for which we have `temp.sml`. Note that `temp.sml` is also used to represent our infinite register in Intermediate Representation.

## 4.3 Finding Escaped Variables

### Associated File(s) Link

`findescape.sml`

*Please see comments in the file(s) for more details.*

It would be important for us to determine which variable cannot go to a register, i.e., which variable escapes. Idea of finding escapes is straight forward: Just see if this thing was defined in outer scope, note that scope extends only when we are calling a function. Escaping is calculated in `findescape.sml`

## 4.4 Intermediate Tree Representation

### Associated File(s) Link

tree.sml

tree.sig

*Please see comments in the file(s) for more details.*

The files `tree.sml`, `tree.sig` consists of datatype representing our Intermediate Tree. Description of which is presented as below.

```
datatype stm = SEQ of stm * stm (* The statement s2 followed by s2. *)
| LABEL of label (* Define the constant value of name n to be the
current machine code address. This is like a label definition in
assembly language. *)
| JUMP of exp * label list (* Transfer control (jump) to address exp.
The destination exp may be a literal label, as in NAME(lab), or it may
be an address calculated by any other kind of expression. For example,
a C-language switch (i) statement may be implemented by doing
arithmetic on i. The list of labels labs specifies all the possible
locations that the expression exp can evaluate to; this is necessary
for dataflow analysis later. The common case of jumping to a known
label "l" is written as jump(name l, [l]). *)
| CJUMP of relop * exp * exp * label * label (* CJUMP(o, e1, e2, t, f):
Evaluate e1, e2 in that order, yielding values a, b. Then compare a, b
using the relational operator o. If the result is true, jump to t
otherwise jump to f. The relational operators *)
| MOVE of exp * exp
(*
MOVE(TEMP t, e): Evaluate e and move it into temporary t.
```

*MOVE(MEM(e1), e2) Evaluate e1 yielding address a. Then evaluate e2 and store the result into wordSize bytes of memory starting at a.*

*\*)*

| **EXP** of exp (*\* Evaluate exp and discard the result. \**)

and **exp** = **BINOP** of binop \* exp \* exp (*\* The application of binary operators to operands exp1, exp2. \**)

| **MEM** of exp (*\* The contents of wordSize bytes of memory starting at address exp (where wordSize is defined in the Frame module). Note that when MEM is used as the left child of a move, it means "store," but anywhere else it means "fetch." \**)

| **TEMP** of Temp.temp (*\* Temporary t. A temporary in the abstract machine is similar to a register in a real machine. However, the abstract machine has an infinite number of temporaries. \**)

| **ESEQ** of stm \* exp (*\* The statement s is evaluated for side effects, then e is evaluated for result \**)

| **NAME** of label (*\* The value NAME(n) may be the target of jumps, calls, etc. \**)

| **CONST** of int (*\* The integer constant int. \**)

| **CALL** of exp \* exp list (*\* A procedure call: the application of function exp1 to argument list exp2 list. The subexpression exp1 is evaluated before the arguments which are evaluated left to right. \**)

and **binop** = **PLUS** | **MINUS** | **MUL** | **DIV** | **AND** | **OR** | **LSHIFT** | **RSHIFT** | **ARSHIFT** | **XOR**

and **relop** = **EQ** | **NE** | **LT** | **GT** | **LE** | **GE** | **ULT** | **ULE** | **UGT** | **UGE**



## 4.5 Understanding Functions Calls in Tiger

Note that some of this material will become clear after reading Phase III.

Below is explained in chronological sequence of what happens and for what reason when function call is executed in this compiler.

1. When a function is called. The current frame is extended to include outgoing parameters (at the offset already determined by the callee) in case some of them escape; rest of the arguments are put in argument registers.
  1. This step is done by our code generator.
  2. After moving escaped arguments to their pre-determined location and remaining arguments to argument registers, it will then emit `jal` instruction which will have `src = argTemps` (`argTemps` are chosen argument registers, this is done as these argument registers are used in this time) and `dst = F.getFirstL F.callersaves`; as we know that caller is supposed to save some registers if it was using them and thus setting `dst = F.getFirstL F.callersaves` would enable the garbage collection to know that if function being called **uses** these registers then it would clearly interfere.
  3. Let `fp`, `sp` denote the current frame and stack pointer.
  4. To extend the current frame, we must subtract it by the amount `escaped-arguments * word-size`. But since we should as well store the old frame pointer (as we will update it with the current stack pointer); we must subtract `sp` by `(escaped-arguments + 1) * word-size`.
  5. Now old value of `fp` is saved in 0th location of this `sp`, and other arguments are saved respectively at the offsets already determined by the callee. (Callee predetermined these offsets considering the fact that we will save old `fp` at 0th word)

6. Now as we are moving to the frame of other function **fp** is updated to **sp**, and **sp** value will be deducted by the amount needed by the callee stack.
  7. Thus local variables allocated will be referred with negative offset wrt to **fp** and escaped argument parameters will be referred with non negative offset wrt **fp**.
  8. Thus it is evident that our code generator would need to know the access list of the callee which wasn't there in the design mentioned in the book, so I added it in **tree.sml** but to avoid cyclic dependency between **tree.sml** and **riscframe.sml**, I had to redefine **access** and had to create **accessConv.sml** to facilitate conversion between access of **tree** and **riscframe**.
  9. Also since code generator must save escaped arguments with respect to the new frame pointer which is not equal to current frame pointer as this whole thing is updated in **procEntryExit3**, but since new frame pointer = current stack pointer - (escape-count + 1) \* word-size, I created new function **callexp** in **riscframe** to do this arithmetic.
2. Now inside this called function, we must create new temporary for each passed argument **in argument register** and move that register's value to this new temporary. This might seem unnecessary but consider **function m(x : int, y : int) = (h(y, y); h(x, x))**. If **x** stays in "parameter register 1" throughout **m**, and **y** is passed to **h** in parameter register 1, then there is a problem. The register allocator will eventually choose which machine register should hold the temporary. If there is no interference of the type shown in function **m**, then (on the RISC) the allocator will take care to choose register the same register as temporary to hold that register (not implemented as of now). Then the move instructions will be unnecessary and will be deleted at that time.
1. This is done in **newFrame** of **riscframe**, named as **shiftInstr** (shift instructions).

3. Called function must save callee-save registers include `ra`, this is done in `procEntryExit1`, similarly restoring them in the end of function body is as well done here.

## 4.6 RISC Frame

### Associated File(s) Link

`riscframe.sml`

*Please see comments in the file(s) for more details.*

File `riscframe.sml` contains an interface for RISC V, which finds some use in this phase, but is mainly there for code generation phase. Its applications and purpose will become clear by seeing the written code. Before proceeding please read and understand about **static links** given at page 132-134 of the *Tiger Book*. Basic gist is that in languages that allow nested function declarations, the inner functions may use variables declared in outer functions. One way to accomplish this is that whenever a function  $f$  is called, it can be passed a pointer to the frame of the function statically enclosing  $f$ ; this pointer is the static link.

## 4.7 Translating to IR

### Associated File(s) Link

`translate.sml`

*Please see comments in the file(s) for more details.*

Done by `translate.sml` and is used by `semant.sml` to generate IR tree code.

Its not reasonable to translate expression of our AST to an expression of IR as this is true only for certain kinds of expressions, the ones that compute a value. Expressions that return no value (such as some procedure calls, or while expressions in the Tiger language) are more naturally represented by `Tree.stm`. And expressions with Boolean values, such as  $a > b$ , might best be represented as a conditional jump - a combination of `Tree.stm` and

a pair of destinations represented by `Temp.labels`. Therefore, we will make a datatype `exp` in the `Translate` module to model these three kinds of expressions:

```
datatype exp = Ex of Tr.exp
             | Nx of Tr.stm
             | Cx of Temp.label * Temp.label -> Tr.stm
```

- **Ex** stands for an "expression," represented as a `Tree.exp`.
- **Nx** stands for "no result," represented as a `Tree.statement`.
- **Cx** stands for "conditional," represented as a function from label-pair to statement.

If you pass it a true-destination and a false-destination, it will make a statement that evaluates some conditionals and then jumps to one of the destinations (the statement will never "fall through"). For example, the Tiger expression

$$a > b \mid c < d$$

might translate to the conditional:

```
Cx(fn (t,f) => SEQ(CJUMP(GT, a, b, t, z),
SEQ(LABEL z, CJUMP (LT, c, d, t, f))))
```

for some new label `z`.

Sometimes we will have an expression of one kind and we will need to convert it to an equivalent expression of another kind. For example, the Tiger statement

```
flag := (a>b | c<d)
```

. This is achieved by `unEx`. Similarly we have `unNx` and `unCx`.

Similarly each function call defines a level which is encapsulated as

```
datatype level = Top
              | Lev of {parent: level, frame: F.frame} * unit ref
```

`unit ref` is used to easily check for equality.

All of the functions of `translate.sml` are straight forward so please see and understand them.

## 4.8 Semantic Analysis

Associated File(s) Link
<a href="#">semant.sml</a>
<i>Please see comments in the file(s) for more details.</i>

After we receive our AST, we run semant's (in `semant.sml`) function `transProg` on it which will do type checking of our AST and simultaneously convert it into our IR. Both this and `translate.sml` are big so I have not put their code here, however please see linked code which is properly documented and thus is easy to understand.



# Chapter 5

## Phase III : Generating Assembly Code

This final phase generates the machine assembly code and thus completing the purpose of compiler. It involves the discussion of various concepts such as Canonisation, Instruction Selection, Liveness Analysis and Register Allocation.

Each of these is explained in an understandable order below.

### 5.1 Canonisation

Associated File(s) Link

canon.sml

*Please see comments in the file(s) for more details.*

#### 5.1.1 Abstract

It's useful to be able to evaluate the sub-expressions of an expression in any order. If tree expressions did not contain ESEQ and CALL nodes, then the order of evaluation would not matter.

### 5.1.2 Why CALL nodes are an issue?

In actual implementation, CALL nodes will return value in the same register (a0 in case of RISC V). Thus in an expression like BINOP(PLUS, CALL(...), CALL(...)); the second call will overwrite the a0 register before the PLUS can be executed.

Remedy is to do the transformation; CALL(fun, args) -> ESEQ(MOVE(TEMP t, CALL(fun, args)), TEMP t)

### 5.1.3 Why ESEQ nodes are an issue?

Clearly in case of simple ESEQ(s, e), statement s can have direct or side effects on an expression e.

Remedy is as shown in figure 5.1 (basically lifting them higher and higher until they become SEQ nodes).

The transformation is done in three stages: First, a tree is rewritten into a list of canonical trees without SEQ or ESEQ nodes; then this list is grouped into a set of basic blocks, which contain no internal jumps or labels; then the basic blocks are ordered into a set of traces in which every CJUMP is immediately followed by its false label. This will become clear when seeing the well documented code.

## 5.2 Instruction Selection

### Associated File(s) Link

[assem.sml](#)

[codegen.sml](#)

[accessConv.sml](#)

*Please see comments in the file(s) for more details.*



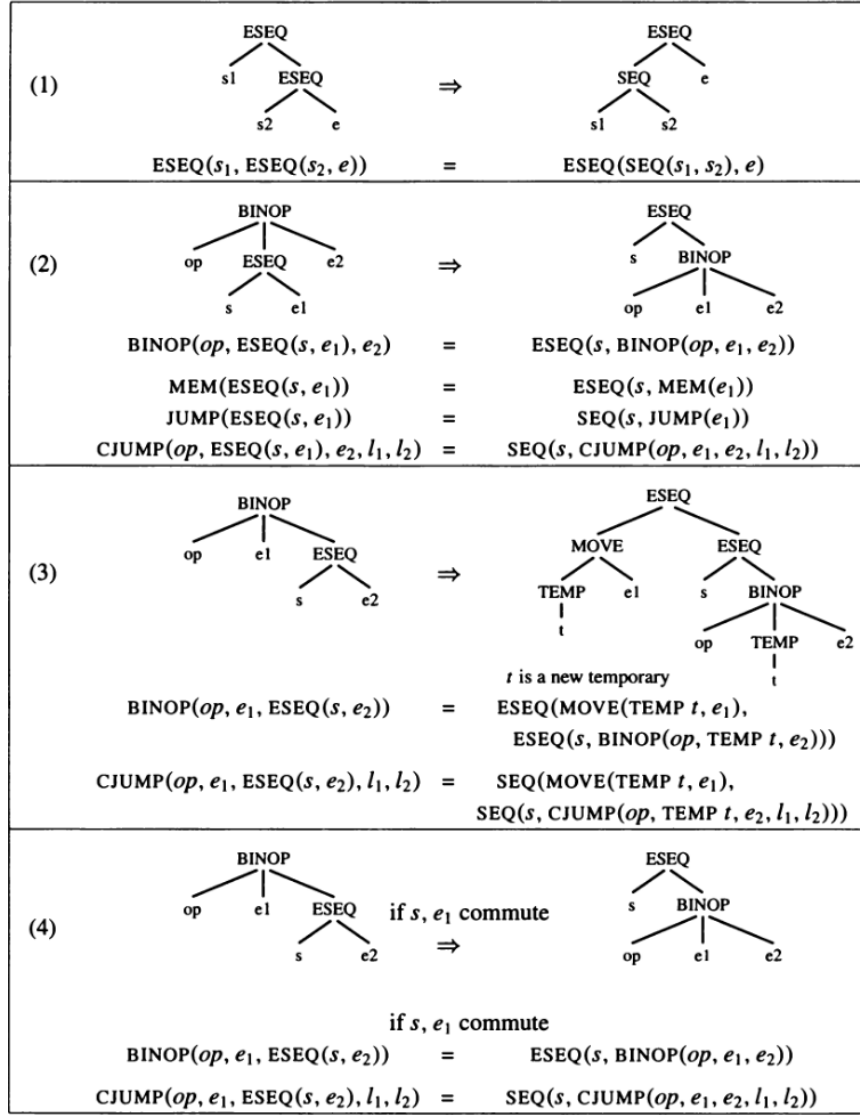


Fig. 5.1: ESEQ Removal. Image source: [1]

### 5.2.1 Instruction Representation

Now that we have done Canonisation, our compiler will now call code generator. It will basically convert the body into assembly language code with the restriction that we will still be using temporaries instead of actual machine registers. In this form, source registers will be labelled ``si` and destination registers as ``di`. So our each instruction would be represented as an operation string having these ``si`, ``di`'s where ``si` is indexed from `src` list, and ``di` are indexed from `dst` list. This is captured as datatype `instr` in `assem.sml`.

```
datatype instr =  
    OPER of {assem: string, dst: temp list, src: temp list, jump:  
            label list option}  
  | LABEL of {assem: string, lab: Temp.label}  
  | MOVE of {assem: string, dst: temp, src: temp}
```

An `OPER` holds an assembly-language instruction `assem`, a list of operand registers `src`, and a list of result registers `dst`. Either of these lists may be empty. Operations that always fall through to the next instruction have `jump = NONE`; other operations have a list of “target” labels to which they may jump (this list must explicitly include the next instruction if it is possible to fall through to it).

A `LABEL` is a point in a program to which jumps may go. It has an `assem` component showing how the label will look in the assembly-language program, and a `lab` component identifying which label-symbol was used.

A `MOVE` is like an `OPER`, but must perform only data transfer. Then, if the `dst` and `src` temporaries are assigned to the same register, the `MOVE` can later be deleted (as done in my register allocator).

### 5.2.2 Code Generation

Finding the appropriate machine instructions to implement a given intermediate representation tree is the job of the instruction selection phase of a compiler.

We can express a machine instruction as a fragment of an IR tree, called a tree pattern as denoted in the figure below. Then instruction selection becomes the task of tiling the tree with a minimal set of tree patterns.

Name	Effect	Trees
—	$r_i$	TEMP
ADD	$r_i \leftarrow r_j + r_k$	
MUL	$r_i \leftarrow r_j \times r_k$	
SUB	$r_i \leftarrow r_j - r_k$	
DIV	$r_i \leftarrow r_j / r_k$	
ADDI	$r_i \leftarrow r_j + c$	
SUBI	$r_i \leftarrow r_j - c$	
LOAD	$r_i \leftarrow M[r_j + c]$	
STORE	$M[r_j + c] \leftarrow r_i$	
MOVEM	$M[r_j] \leftarrow M[r_i]$	

**Fig. 5.2:** Tree Patterns showing arithmetic and memory instructions. Image source: [1]

The very first entry is not really an instruction, but expresses the idea that a **TEMP** node is implemented as a register, so it can “produce a result in a register” without executing any instructions at all.

For each instruction, the tree-patterns it implements are shown. Some instructions correspond to more than one tree pattern; the alternate patterns are obtained for commutative operators (+ and \*), and in some cases where a register or constant can be zero (**LOAD** and **STORE**). Here we abbreviate the tree diagrams slightly: **BINOP**(**PLUS**, **x**, **y**) nodes will be

written as  $+(x, y)$ , and the actual values of CONST and TEMP nodes will not always be shown.

Consider the instruction  $a[i] := x$ . Two possible tilings for this instruction is shown in the diagram. (Remember that  $a$  is really the frame offset of the pointer to an array)

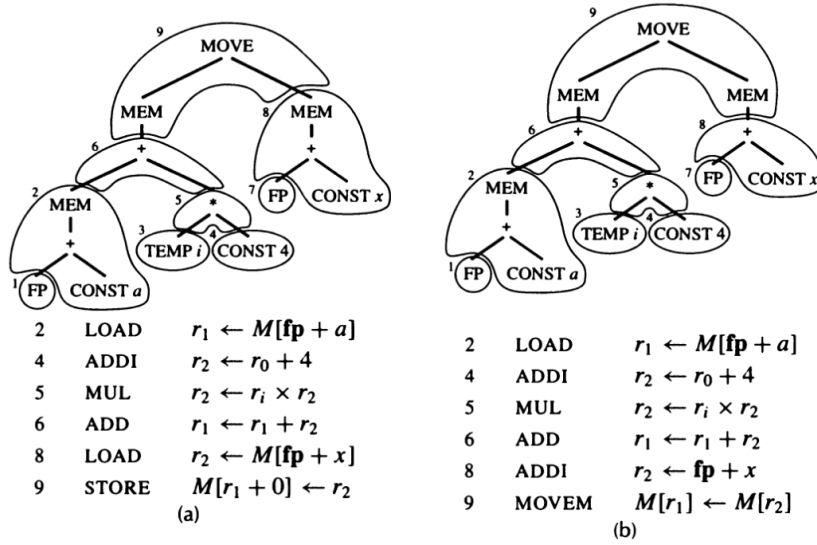


Fig. 5.3: A tree tiled in two ways. Image source: [1]

### 5.2.3 Maximal Munch Algorithm

Suppose we could give each kind of instruction a cost. Then we could define an optimum tiling as the one whose tiles sum to the lowest possible value. An optimal tiling is one where no two adjacent tiles can be combined into a single tile of lower cost. If there is some tree pattern that can be split into several tiles of lower combined cost, then we should remove that pattern from our catalog of tiles before we begin. Every optimum tiling is also optimal, but not vice versa.

The algorithm for optimal tiling is called **Maximal Munch**. It is quite simple. Starting at the root of the tree, find the largest tile that fits. Cover the root node - and perhaps several other nodes near the root - with this tile, leaving several subtrees. Now repeat the same algorithm for each subtree. As each tile is placed, the corresponding instruction is

generated. The Maximal Munch algorithm generates the instructions in reverse order - after all, the instruction at the root is the first to be generated, but it can only execute after the other instructions have produced operand values in registers. The “largest tile” is the one with the most nodes. If two tiles of equal size match at the root, then the choice between them is arbitrary. Maximal Munch is quite straightforward to implement in ML. Simply write two recursive functions, `munchStm` for statements and `munchExp` for expressions. Each clause of `munchExp` will match one tile. The clauses are ordered in order of tile preference (biggest tiles first); ML’s pattern-matching always chooses the first rule that matches.

This algorithm is implemented in the file `codegen.sml`

## 5.3 Liveness Analysis and Interference Graph

### Associated File(s) Link

[flowgraph.sml](#)

[makegraph.sml](#)

[liveness.sml](#)

*Please see comments in the file(s) for more details.*

### 5.3.1 Liveness Analysis

Two temporaries **a** and **b** can fit into the same register, if **a** and **b** are never “in use” at the same time.

We say a variable is live if it holds a value that may be needed in the future, so this analysis is called liveness analysis. To perform analyses on a program, it is often useful to make a control-flow graph. Each statement in the program is a node in the flow graph; if statement **x** can be followed by statement **y** there is an edge from **x** to **y**.

In this compiler, flow graph is as represented in `flowgraph.sml`.

Similarly `makegraph.sml` constructs this graph.

A variable is live on an edge if there is a directed path from that edge to a use of the variable that does not go through any `def`. A variable is live-in at a node if it is live on any of the in-edges of that node; it is live-out at a node if it is live on any of the out-edges of the node.

Liveness of node is calculated as shown<sup>[1]</sup>:

$$in[n] = use[n] \cup (out[n] \setminus def[n])$$

$$out[n] = \cup_{s \in succ[n]} in[s]$$

A condition that prevents `a` and `b` being allocated to the same register is called an interference.

The most common kind of interference is caused by overlapping live ranges when `a` and `b` are both live at the same program point, then they cannot be put in the same register.

### 5.3.2 Interference Graph

Interference graph is created with vertices as temporaries and edges as follows<sup>[1]</sup>:-

1. At any nonmove instruction that defines a variable  $a$ , where the live-out variables are  $b_1, \dots, b_j$ , add interference edges  $(a, b_1), \dots, (a, b_j)$ .
2. At a move instruction  $a \leftarrow c$ , where variables  $b_1, \dots, b_j$  are live-out, add interference edges  $(a, b_1), \dots, (a, b_j)$  for any  $b_i$  that is not the same as  $c$ .

---

All this is handled in `liveness.sml`

## 5.4 Register Allocation

### Associated File(s) Link

regalloc.sml

regalloc.sig

*Please see comments in the file(s) for more details.*

Now that we have made our interference graph, where each edge denotes that its end-points must go in different registers, thus, our problem reduced into coloring our graph with  $K$  (# no. of available registers) colors such that no two end points have same colour.

As of now this is implemented by below mentioned algorithm **but** later I would implement the exact algorithm which is given in book. Algorithm is implemented in the file `regalloc.sml` and `regalloc.sig`

Note that graph coloring problem is not fixed parameter tractable with respect to number of colors.

### 5.4.1 Algorithm

Recursive algorithm is defined as below, assume that we are given a set of instructions each of which may contain temporaries which might have been already mapped to some register.

1. Construct interference graph given the list of instructions.
2. Start with vertices (temporaries) which have not been assigned a colour (register).
3. Such vertices whose  $\# \text{ Neighbours} < K$  can be removed from the graph as we would always have a color available for them. Thus this divides our list of vertices into two viz., (`simplify`, `potentialSpill`).
4. Now we should process these nodes in `simplify`, let “`stack`” denote the set of simplified nodes.

5. Repeat until `potentialSpill` nodes become empty
  1. We “simplify” each node in our `simplify` list by removing it from `simplify` list and adding it to `stack` and decrementing the degree of its neighbors which are not in `stack` (`stack` nodes are already removed) and also which aren’t already colored. Note that when we decrement the degree of the neighbors, if degree becomes less than  $K$  that means we can add this node to `simplify` and remove it from `potentialSpill`.
  2. If `potentialSpill` is empty then done, o/w select a node to spill, choose that which has maximum neighbors and should have accessed temps/memory least number of times. Assign it to `simplify` and remove it from `potentialSpill`.
6. Now process `stack` from top to bottom. For an element, colors available to it are total minus those taken by its neighbors, if colors are available then assign else we add it to `actualSpill`.
7. If by now `actualSpill` is empty then we stop else we rewrite the program and repeat.



# References

- [1] A. W. Appel, *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.
- [2] J. S. M. Andrew W. Appel and D. R. Tarditi. User's guide to ml-lex and ml-yacc. [Online]. Available: <http://www.cs.tufts.edu/comp/181/ug.pdf>
- [3] B. Landers. Rars wiki. [Online]. Available: <https://github.com/TheThirdOne/rars/wiki>