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

List of Figures

Chapter 1

Introduction

This project aims to write a compiler to compiler Tiger to RISC V assembly code.

Note

Before proceeding it is important to read about Tiger language from [1] *(read Tiger Reference Manual in Appendix)*.

Beyond the specification of the language mentioned within text [1], my Tiger specification includes abstractly two more things:-

1. Real number support.

2. Objective functionality support.

Real numbers enjoy first class support just like integers, strings and can be used exactly as integers.

1.1 Grammar Additions

$exp \rightarrow$ *real-literal*

$\rightarrow$ ***new*** *class-id*

$\rightarrow$ *lvalue.id()*

$\rightarrow$ *lvalue.id(exp*$\{$*, exp*$\}$*)*

dec $\rightarrow$ *classdec*

classdec $\rightarrow$ ***class*** *class-id* ***extends*** *class-ids* $\{\{$*classfield*$\}\}$

class-ids $\rightarrow$ *id*$\{$*, id*$\}$

classfield $\rightarrow$ *vardec*

classfield $\rightarrow$ *fundecs*

1.2 Specifications For Objective Tiger

For the purpose of understanding the below points, refer 1.1

```
let
  var start := 10
  class Vehicle extends Object {
    var position := start
    function move(self : Vehicle, x : int) = (self.position := self.position + x)
  }
  class Car extends Vehicle {
    var passengers := 0
    function await (self : Car, v : Vehicle) =
      if (v.position < self.position) then v.move (self.position - v.position)
      else self.move(10)
  }
  class Truck extends Vehicle {
    var position := start + 5
    function move (self : Truck, x : int) = if x <= 55 then self.position := self.position + x
  }
  var c := new Car
  var t := new Truck
  var v : Vehicle := c
in
  t.move (55); print("Truck after moving 55 units starting from 10: "); printI(t.position); print("\n");
  t.move (56); print("Truck after trying to move 56 (not allowed): "); printI(t.position); print("\n");
  c.move(100); print ("Car after moving 100 units starting from 10: "); printI(c.position); print("\n");
  c.await(t); print ("Car awaiting for truck, now truck's position is: "); printI(t.position); print("\n");
  print("Car passengers: "); printI(c.passengers); print("\n");
  c.passengers := 2;
  print ("Car passengers: "); printI(c.passengers); print("\n");
  if ISTYPE(c, "Vehicle") then print("Car is a Vehicle too!\n"); exit(0)
end
```

Fig. 1.1: Classes Demo I

- To simplify implementation, all declared *classes* (irrespective of being in different environment) should have unique name.
- *vardec*, *fundecs* could be *overridden* by the subclass but the types should be preserved as it has to be backward compatible.
- *vardec*'s within the class shall not use this subclass fields for its initialisation as there is no self to fetch them.

- As evident from grammar specification, there is no "method" but our usual "functions" and thus mutually recursive methods are supported.

- Type casting of a subclass to its superclass is supported both in variable assignment and as in function argument. This is termed as "SUPERTYPE". We can see in line 19 that though variable `c` is of type `Car` yet we have assigned it to a variable of type `Vehicle`. Similarly in line 24, when executing `c.await(t)`, we can very well see that the `await` function expects an object of type `Vehicle` but we have given an object which is a subclass of it.

- There is also a support to check whether an object 'x' is of its original class 'X' or any of the superclass of 'X' via *ISTYPE(objectName, ClassName)*. *ClassName* is to be given as a *string* whereas the other parameter shall be *var*.

- Multiple inheritance is supported but the classes to be extend should be *disjoint* and thus should have no field in common.

- Classes to support solely *vars* and *fundecs* which *could* be redefined as evident by redefinition of variable `position` and function `move` in line 14, 15 respectively.

- Concept of *self*. When calling a function of an object, one need not pass its own discriptor which is done via compiler but when declaring the function of a class, one needs to mention *self* as an argument.

1.3 Compile Time Arguments

Since there are so many choices offered to a user when using this compiler. For instance:-

1. Option to give filename to print IR Tree at. Flag = `ir`. Default = `TextIO.stdOut`.

2. Option to give filename to print instructions before register allocation at. Flag = `ba`. Default = `TextIO.stdOut`.

3. Option to choose which string algorithm to be used for displaying suggestions in case of mistyped word. Flag = `sa`. Default = `0` (i.e. Levenshtein algorithm).

I have implemented feature for users to give compile time flags. Note: I mention "flags", thus user has an option to not define it, in which case default flag will be taken. See 1.2 for an example usage.

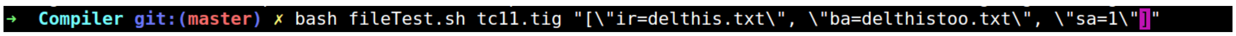


Fig. 1.2: Compile time arguments

1.4 How To Compile?

One needs to first *make* the files visible in *sources.cm* as `CM.make (\sources.cm")` then simply compile the file using the command (in sml), `Main.compile(FilePath, Arguments)`. For an example please see the above subsection and aswell the file `fileTest.sh`, contents of which are listed in 3.3.

One can now run the compiled RISC V code using RARS like `java -jar rars1_3_1.jar sm nc TestFiles/filename.tig.s`

1.5 Organisation of Report

The second chapter explains how to add Objective functionality to Tiger. Third, Fourth chapter briefly explains the work done for this project, i.e. 8^{th} Semester and 7^{th} Semester respectively.

These chapters also explains various functionalities provided by my Compiler. The code written for this compiler is enormous and the remaining part of the report gives documentation on each of these files in an understandable order.

Chapter 2

Adding Objective Support in Tiger

As explained in first Chapter, Classes can have only two type of fields, viz., *vars* and *functions*.

2.1 Support for *vars*

One can understand on how to add support for *vars* provided one understands the working of *record*. Abstractly one can think of Class as being union of *vars* and *functions*. Thus, these two could be *separately* supported. If we forget about *functions* then the Class (besides other Class functionalities like Multiple Inheritance, "SUPERTYPE", etc) simply reduces to a *record*. So *vars* support has been added as its done in case of *records*. Please see `semant.sml` for more details.

2.2 Support for *functions/methods*

Unlike *vars*, which could be "stored", *functions* are "declared". One should understand this as this is pivotal to successfully implement *functions*. Thus, if we are adding Class type to our environment, we shall add its declared *functions* in our environment with *certain* modification. This modification is needed as these functions are not "outside" Class's declaration thus can't be called upon without corresponding Object. One way to achieve this is to mask the functions name (say "F") of Class (say "C") to "Class_C_F". This is how I handle the functions.

2.3 Supporting Single Inheritance

```
class A extends Object {
  var a := "a"
}
class B extends A {
  var b := "b"
  var c := "c"
}
class C extends A {
  var d := "d"
}
class D extends B {
  var e := "e"
}
```

Fig. 2.1: Single Inheritance Program

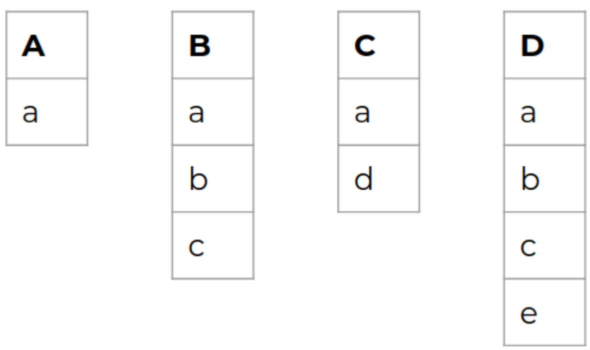


Fig. 2.2: Single Inheritance Fields Allocation

Since we must support backward compatibility, the idea of "SUPERTYPE". Superclass's fields should be listed before the Subclass's fields as shown in figure 2.2 of program 2.1. Since I support for fields redefinition, one should be cautious when implementing this.

2.4 Support for Multiple Inheritance

This would have been easy in case we were to not support "SUPERTYPE". It is the combination of Multiple Inheritance and backward compatibility that makes this problem tough.

To understand this, see 2.3 and 2.4. We can see that we just can't list the fields linearly for any class and inevitably there are gaps left.

```
class A extends Object {
  var a := "a"
}
class B extends Object {
  var b := "b"
  var c := "c"
}
class C extends A {
  var d := "d"
}
class D extends C, B {
  var e := "e"
}
```

Fig. 2.3: Multiple Inheritance Program

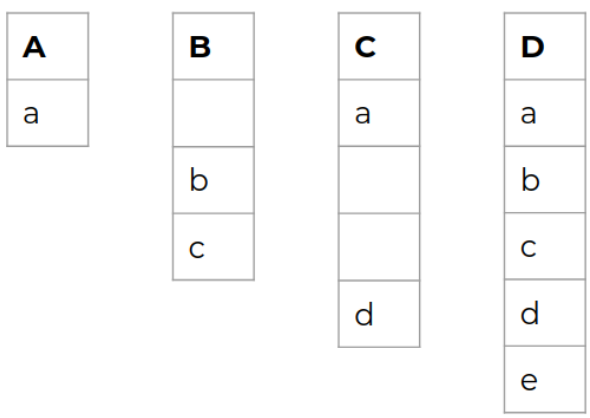


Fig. 2.4: Multiple Inheritance Fields Allocation

To handle this, we can "statically" analyse the complete program and reduce this problem to graph coloring. Idea is to add an edge between *vars* which could possibly end being in the same class. Then we would be ended with disjoint Cliques which could then be colored.

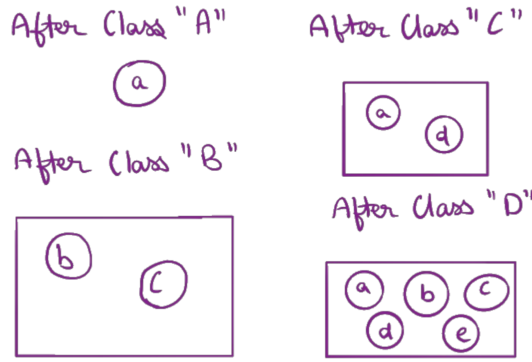


Fig. 2.5: UFDS Example

But we can do better. As the graph is simply disjoint cliques, each node in a component would receive different color. Thus we can simply use *Union Find Disjoint Set* (UFDS) data structure *[Which is practically a constant time algorithm]* and color each element of the set with different color. Here we just need to join the set containing *vars* of Superclasses with Subclass. See 2.5 for an example for the code 2.3.

2.5 Support for *ISTYPE*

With this section, I will also explain on how to support type casting. Basically for type casting we just need to check whether this type is even compatible with given type which would be possible if the Class concerned is Superclass or original class of the given object. So for given class, one should store the list of all classes which it extends. Now to see type casting or type compatibility, we try to match the given class with the classes in this list and continue recursively until we reach the base Object Class.

Now *ISTYPE* uses this procedure to check validity. This is done in `semant.sml` and after determining, I here itself replace this program code with simply its answer which is either integer 0 or 1.

Chapter 3

What Has Been Achieved This Semester

3.1 Miscellaneous Improvements

3.1.1 More Graceful Error Termination

Earlier in lexical phase, as soon as an error was found, I was raising an exception but it could be better presented as simply an exit failure, or even better an exit success with just my simple lexer error. Which reduces the error output received by the user. Figure 3.1 and 3.2 shows the reduction in those "uncaught..." lines.

3.1.2 Simplified "Make" Process

Earlier, the bash script to compile the given file required to concatenate the output with the `runtime.s`. Now it has been significantly simplified and the compilation process is reduced to just `Main.compile filename (optional flags)`. This concatenation business is now handled in the `Main` file itself. See 3.3.

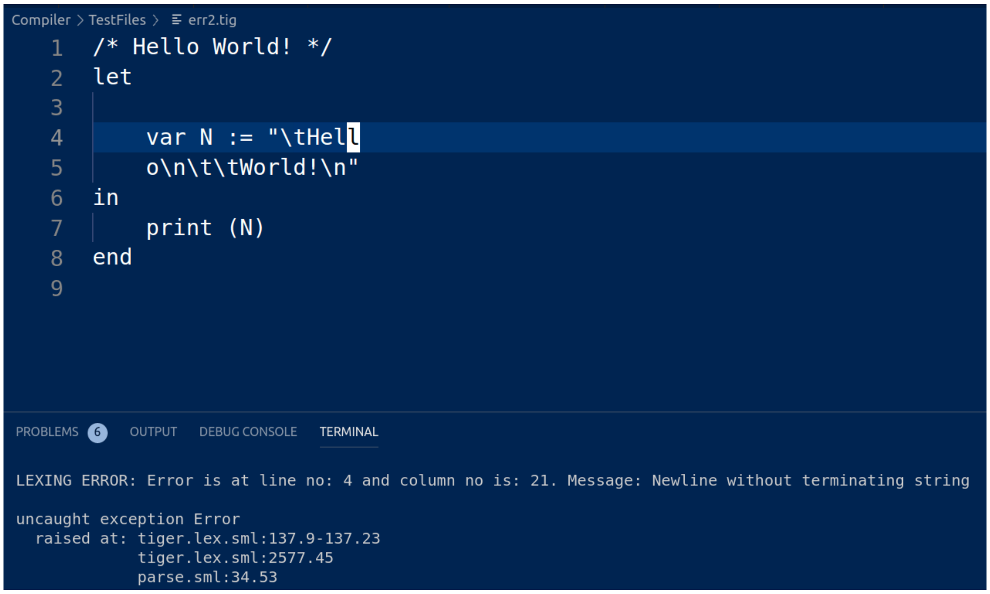


Fig. 3.1: Graceful error termination - before

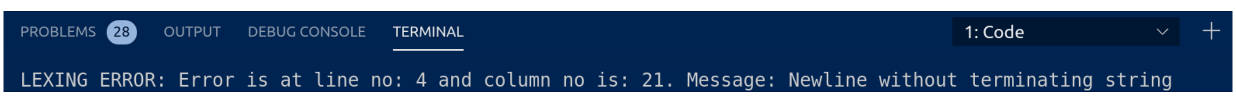


Fig. 3.2: Graceful error termination - after

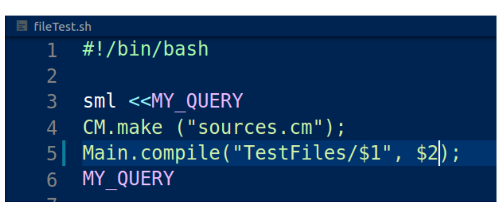


Fig. 3.3: Simplified Make

3.1.3 Drew My Own Tiger Logo

See 3.4.

Fig. 3.4: Tiger Logo

3.1.4 Printing IR Tree As Well As Generated Code Before Register Allocation

The author of the book gives the program to achieve this. But due to changes commited by me in IR Tree, etc. hindered myself from using that code as now it would demand modifications to use but I didn't find much use of it later. Now as I was getting stuck time and again in various phases of these new improvements I was targeting. It became necessary to study IR tree and the code before register allocation to pin point where exactly the error is. See 3.5 and 3.6.

3.1.5 Others

- Added more details in the project's documentation (with more diagrams) to make it more useful for lucid revision. Along with general improvements in code.
- Removed External Calls (this lead to simplification in code generator, semantic, translate and risc frame files).
- `FP` is indispensable, thus I decided not to remove it by calculating it using `SP`.

```
---------- IR Tree ---------
SEQ(
 MOVE(
  TEMP t233,
  TEMP t101),
 SEQ(
  MOVE(
   TEMP t234,
   TEMP t139),
  SEQ(
   MOVE(
    TEMP t235,
    TEMP t140),
   SEQ(
    MOVE(
     TEMP t236,
     TEMP t141),
    SEQ(
     MOVE(
      TEMP t237,
      TEMP t142),
     SEQ(
      MOVE(
       TEMP t238,
       TEMP t143),
      SEQ(
       MOVE(
        TEMP t239,
        TEMP t144),
       SEQ(
        MOVE(
         TEMP t240,
         TEMP t145),
        SEQ(
         MOVE(
          TEMP t241,
```

Fig. 3.5: IR Tree

```
---------- Instr before allocation ---------
L1:
mv (222, 0), (101, 0)
mv (223, 0), (139, 0)
mv (224, 0), (140, 0)
mv (225, 0), (141, 0)
mv (226, 0), (142, 0)
mv (227, 0), (143, 0)
mv (228, 0), (144, 0)
mv (229, 0), (145, 0)
mv (230, 0), (146, 0)
mv (231, 0), (147, 0)
mv (232, 0), (148, 0)
mv (233, 0), (149, 0)
fmv.s (234, 1), (150, 1)
fmv.s (235, 1), (151, 1)
fmv.s (236, 1), (152, 1)
fmv.s (237, 1), (153, 1)
fmv.s (238, 1), (154, 1)
fmv.s (239, 1), (155, 1)
fmv.s (240, 1), (156, 1)
fmv.s (241, 1), (157, 1)
fmv.s (242, 1), (158, 1)
fmv.s (243, 1), (159, 1)
fmv.s (244, 1), (160, 1)
fmv.s (245, 1), (161, 1)
li (103, 0), 0
jal exit
mv (103, 0), (103, 0)
fmv.s (161, 1), (245, 1)
fmv.s (160, 1), (244, 1)
fmv.s (159, 1), (243, 1)
fmv.s (158, 1), (242, 1)
fmv.s (157, 1), (241, 1)
```

Fig. 3.6: Code before register allocation

3.2 Simplified String Comparisons

Comparing strings in a program is indispensable. Therefore, it is inconvinient to have user type a function each time there is a need to call a string function. Thus, I have added this funcionality where string comparisons can be done naturally.

Idea behind its implementation is that whenever lexer sees `str1 > str2`. Then in semantic analysis phase, we should determine whether both sides of an operator are of type strings or not. If that is the case then we can delete this node and replace it with a function call as inequalities are like conditions. So one should have supporting functions defined in `runtime.s` along at other places. See 3.7 for an example usage.

Fig. 3.7: String Comparisons

3.3 Suggestions for Mistyped Words

Tiger's target has always been to continue in case there are any mistakes and show maximum possible mistakes to the user at once.

Keeping this aim in mind, it is as well desired to print suggestions for a mistyped word which could be either a variable/function or a type name.

Earlier I abstracted out `Not Found` errors in semantic phase but failed to proceed from that point in that semester. This semester, I began on continuing from where I left. I did **huge** amount of changes where I completely removed `symbol.sml`, `table.sig` and `table.sml`. I replaced their used everywhere with `atom`. And as this was a huge change, I expected many errors. Ultimately I lost to the errors and as I got the new insights, this was anyway not needed. **Idea** is that, anytime if somebody calls this symbol function, we are storing this new symbol. So if one is anyway storing a symbol, better I can just query this `symbol.sml` to ask whether a string is present in symbol table or not. If it is present I can get a corresponding symbol which I can use to check whether that symbol is present in a given environment. I can even map an integer (denoting string's length) to a set of symbols. Keeping these things in mind, I have implemented two algorithms:-

3.3.1 Algorithm I

In this algorithm, I am using a map, which maps an integer (denoting string's length) to a set of symbols. Now each time, I encounter a "Not Found" error, I get from this map, sets having strings of length 0 or unit distance away from the given string. Now we can use **Levenshtein's** algorithm to check for minimum edit distance of the given string from this set. Time complexity will work out to be: $\mathcal{O}(m * l^2 * log^2(s))$. Where m equals strings in this set, $l = max(|strings|)$ and s equals total symbols in the environment, we have this log factor for this look up and as well as lookup to get sets corresponding to that length, and quadratic factor for minimum edit distance DP algorithm.

3.3.2 Algorithm II

This is comparatively quick algorithm which successfully attempts to find string which are unit distance apart. The idea is that we can easily find a symbol corresponding to a string (if such a symbol exists) from our `symbols.sml` in *log* factor, then, we can simply check whether this symbol exists in our environment in another *log* factor. Now a string can be unit distance apart by a character deletion, or insertion or replacement (I as well implemented swap check function which is very common form of error). Trying all these possibilites, time complexity works out to be: $\mathcal{O}(log(s) * l * |\Sigma|)$ where s is as defined before, l equals length of the given string and Σ equals set of characters which are to be tried for replacement (this although can be treated as a constant and removed from this asymptotics).

Both the algorithms can have their advantage for certain test cases, it is left to the user which algorithm to use with the help of the flag `sa` as mentioned before. Algorithm 1 is better when there are less number of symbols and user is interested in just the minimum possible edit distance away string which needn't be just one character away.

Implementation of these algorithms can be found in `symbol.sml`.

3.4 Real (Float) Implementation - Phase I

Implementing real number support is a huge challenge as it requires A-Z modifications and in this 4000+ lines of code, even one bug could be fatal and difficult to detect thus making it important to identify at which phase is the error coming. Is it coming in IR generation phase or Code generation phase. Keeping this in view, I augmented `printtree.sml` file given by the author to streamline it with the modifications in my IR tree. There were lots of changes required which I will mention in the phase II of this.

Reals have dual behavior. They behave partly as strings and partly as integers. One cannot simply load a floating point register as in case of integers like `li a1, someInteger`. They need to be first declared at top in `.data` field just like strings, then their address loaded again just like strings, then finally they have to be loaded to a floating point register in a different manner like `flw fa1, address`.

In Tiger specification, each time a function returns a value, it is expected to return an integer value. Similarly, everywhere it is assumed that `temp` will be assigned an integer value. Changing nature of `temps` and full fledged support of reals would imply change at almost all files, which is cumbersome. One can get away from this by treating floats like strings, ofcourse one would require to make changes in all the phases before code generation with dedicated type for floats but treating floats just like strings, would mean that we would always have to play with just address which is an integer and not with any real value as such. One can levy the pain to pre defined functions for performing arithmetic and comparison operations with the drawback that they will return result, just like in case of strings, an address of a newly allocated memory. This is the only drawback of this as one mainly tries to avoid memory accesses. This phase however implements this as it forms a basis for phase II in which I provide complete support for floats.

This phase in itself as well demanded lots of modifications which was not easy. All the extra functions written in `runtime.s` were removed in phase II.

Example of changes required in this phase are given in figure 3.8 and 3.9.

3.5 Register Allocator

My last sem's compiler's register allocator was incomplete and a crude implementation of the algorithm given in the text. It was without coalescing which is essential to minimize the spills and different register's use (in move instruction). The book has a whole chapter dedicated to this subject. While implementing this algorithm, for purpose of optimizations, I did modifications among other files as well list using set everywhere from which one can easily add / delete an entry.

The algorithm given in the book is huge and requires careful implementation. After the algorithm's implementation there were major issues I was facing due to some errors.

After carefully generating small examples which break the code, I realised that `ra` register isn't getting backed up. Error was at incorrectly setting up `dst` registers when calling a function, here I missed to include `ra` register due to which it wasn't causing an interference with its use. See 3.10.

Similarly other errors like inserting in stack in opposite direction, etc, took much time.

But after fixing all this, there was still an unexpected behaviour as the algorithm presented by the author missed an important line. Refer 3.11 where you can see that checking

```
+ radd:
+     flw fa0, (a0)
+     flw fa1, (a1)
+     li a0, 4
+     li a7, 9
+     ecall
+     fadd.s fa0, fa0, fa1
+     fsw fa0, (a0)
+     jr ra
+
+ rsub:
+     flw fa0, (a0)
+     flw fa1, (a1)
+     li a0, 4
+     li a7, 9
+     ecall
+     fsub.s fa0, fa0, fa1
+     fsw fa0, (a0)
+     jr ra
+
+ rmul:
+     flw fa0, (a0)
+     flw fa1, (a1)
+     li a0, 4
+     li a7, 9
+     ecall
+     fmul.s fa0, fa0, fa1
+     fsw fa0, (a0)
+     jr ra
+
+ rdiv:
+     flw fa0, (a0)
+     flw fa1, (a1)
+     li a0, 4
+     li a7, 9
+     ecall
+     fdiv.s fa0, fa0, fa1
+     fsw fa0, (a0)
+     jr ra
```

Fig. 3.8: Real I Implementation Glimpse (a)

```
+       callRealFunction (left, oper, right, pos) =
+       case oper of
+         A.EqOp => trexp(A.CallExp({func = Symbol.symbol "realEqual", args = [left, right], pos = pos}))
+       | A.NeqOp => trexp(A.IfExp({test = A.CallExp({func = Symbol.symbol "realEqual", args = [left, right], pos = pos}
+       | A.LtOp => trexp(A.CallExp({func = Symbol.symbol "realLess", args = [left, right], pos = pos}))
+       | A.LeOp => trexp(A.IfExp({test = A.CallExp({func = Symbol.symbol "realLess", args = [left, right], pos = pos}),
+       | A.GtOp => trexp(A.CallExp({func = Symbol.symbol "realGreat", args = [left, right], pos = pos}))
+       | A.GeOp => trexp(A.IfExp({test = A.CallExp({func = Symbol.symbol "realGreat", args = [left, right], pos = pos}),
+       | A.PlusOp => trexp(A.CallExp({func = Symbol.symbol "radd", args = [left, right], pos = pos}))
+       | A.MinusOp => trexp(A.CallExp({func = Symbol.symbol "rsub", args = [left, right], pos = pos}))
+       | A.TimesOp => trexp(A.CallExp({func = Symbol.symbol "rmul", args = [left, right], pos = pos}))
+       | A.DivideOp => trexp(A.CallExp({func = Symbol.symbol "rdiv", args = [left, right], pos = pos}))
      and
          (* trexp recurs over Absyn.exp and trvar recurs over Absyn.var *)
          (* In rare cases when trexp wants to change venv, it must call transExp instead of just trexp *)
          trexp (A.VarExp(var)) = trvar var
        | trexp (A.NilExp) = {exp = L.nilexp, ty = T.NIL}
        | trexp (A.IntExp(intvalue)) = {exp = L.intlit(intvalue), ty = T.INT}
        | trexp (A.StringExp(stringvalue, pos)) = {exp = L.strlit(stringvalue), ty = T.STRING}
+       | trexp (A.RealExp(realvalue)) = {exp = L.reallit(realvalue), ty = T.REAL}
```

Fig. 3.9: Real I Implementation Glimpse (b)

```
(emit(A.OPER {
    assem = "jal " ^ S.name(label) ^ "\n",
    src = argTemps,
    (* All these dst registers are named, i.e. they are not arbitrary temporaries but are already mapped to
    actual machine regs, thus if the function being called wants to use them, our register allocation will
    handle it. Also see highlighted portion of page 237. *)
    dst = F.getFirstL (F.calldefs),
    jump = NONE});
F.rv)
```

Fig. 3.10: Bug in Register Allocator

whether v is in the precolored nodes or not was critical as precolored nodes should not be touched.

```
FreezeMoves(u) =
TPS.app (
    fn (m as (x, y)) =>
    let
        val v = if (GetAlias (y) = GetAlias (u))
        then (GetAlias (x)) else (GetAlias (y))
    in
    (
        activeMoves := TPS.subtract (!activeMoves, m)
        ;
        if (TPS.isEmpty(NodeMoves (v)) andalso
        getTMIntD(!degree, v) < getK(v) andalso
        TS.member(precolored, v) = false) then (  (*
        adding v check in precolored was pivotal *)
            freezeWorklist := TS.subtract
            (!freezeWorklist, v);
            simplifyWorklist := TS.add
            (!simplifyWorklist, v)
        ) else ()
    )
    end
) (NodeMoves(u))
```

```
    freezeWorklist ← freezeWorklist \ {u}
    simplifyWorklist ← simplifyWorklist ∪ {u}
    FreezeMoves(u)

procedure FreezeMoves(u)
    forall m(=copy(x,y)) ∈ NodeMoves(u)
        if GetAlias(y)=GetAlias(u) then
            v ← GetAlias(x)
        else
            v ← GetAlias(y)
        activeMoves ← activeMoves \ {m}
        frozenMoves ← frozenMoves ∪ {m}
        if NodeMoves(v) = {} ∧ degree[v] < K then
            freezeWorklist ← freezeWorklist \ {v}
            simplifyWorklist ← simplifyWorklist ∪ {v}

procedure SelectSpill()
```

Fig. 3.11: Error in Book's Algorithm

Final generated code was significantly lower than earlier, demonstrating the power of coalescing.

3.6 Real (Float) Implementation - Phase II

As remarked, this phase attempted to provide full fledged real numbers support. Huge number of modifications were needed like:-

1. Everywhere when doing `T.newtemp` one need to **specify** whether, real or an integer register is needed.

2. To provide this information, in places one needs to know that whether called function arguments are real or not. And also whether a function returns a real value or not.

   (a) This sparked changes in `riscframe.sml`, `semant.sml`, `translate.sml`, `tree.sml` and many other files.

   (b) Even in canonisation phase, one needs to generate a new temporary for shifting function's return value, thus for this reason as well one need IR tree to contain information about functions return value.

3. Assign statement needed to know whether parameters are real or not as then only one can do `RMOVE` or an integer move, `MOVE`. This is needed as floating points are moved differently.

4. Similarly floating points are loaded and stored using different instructions than integer, this demanded for addition of `RMEM`.

5. Code generation was thus heavily modified as it needed to generate correct code for all these changes. Attempt was made for those possiblities which require least changes in IR tree.

6. The way code was generated when calling a function was as well heavily modified as one needs to differentiate a real and non real value as their moves, memory access is different. In that case also one needs to identify whether an access is in register or memory.

7. New frame generation in `riscframe.sml` was heavily modified as now shift instructions needed to take care for both ints and real temporaries and also access had to specifically generated according to argument type.

8. All in all there were many changes in almost all the files including the changes which were done previously as now reals can not be treated simply like strings, thus their basic expression had to be changed and this change has to be thoughtful as it seemingly could be done in many ways, so one has to really go through required files to determine what should be the most suitable instruction. This one line (line 117), as seen in 3.12 might seem trivial once it is written but there is a big background behind it.

9. Clearly `temp.sml`, `temp.sig` required changes to support for new temporaries. Which is a tuple now, where first parameter is what was originally the temporary and second parameter is simply an integer, denoting whether the temporary is real or not. 1 for real and 0 for normal integer. Some other functions were added such as whether temporary is real or not, etc.

10. There was a constant shuffle between compiler and assembly code, to determine where exactly were the bugs coming from.

11. Register allocator too had to be cleverly modified. Like making `K` which is number of available colours a tuple as for different registers, this `K` is different. Similarly degree map should be change to give a tuple, telling which are non real, real neighbours respectively. These helped in passing of few test cases but it eventually stumbled. Finally it can be seen that there is no point in adding an edge to a non real and a real temporary as they don't interfere. This solved my issue, and floating point implementation was complete. See 3.13 for an example.

```
fun reallit (r : real) : exp =
let
  val there = List.find
    (fn (fragment) =>
      case fragment of
        F.PROC _ => false
      | F.STRING _ => false
      | F.REAL(_, r') => Real.== (r, r')) (!fragments)
in
  case there of
      NONE =>
      let
        val l = Temp.newlabel()
      in
        (fragments := F.REAL(l, r) :: !fragments; Ex(Tr.RMEM (Tr.NAME(l))))
      end
    | SOME(F.REAL(lab, _)) => Ex(Tr.RMEM (Tr.NAME(lab)))
end
```

Fig. 3.12: Change for floating point

```
let
  var A := 2.12
  var B := 2.33
  var a := A > B
  var b := B > A
  var c := B >= B
  var d := B >= A
  var e := A = A
  var f := B <> A
  var g := A < B
  var h := B < A
  var i := B <= B
  var j := B <= A
  var k := A + B
  var l := A - B
  var m := A * B
  var n := A / B
in
  (printR (A); print ("\n"); printR(B); print("\n"); printI (a); printI(b); printI(c); printI(d); printI(e); printI(f);
  printI(g); printI(h); printI(i); printI(j); print ("\n"); printR (k); print ("\n"); printR (l); print ("\n"); printR
  (m); print ("\n"); printR (n); print ("\n"); exit(0))
```

PROBLEMS 42   OUTPUT   DEBUG CONSOLE   TERMINAL   1: Code

```
-
2.12
2.33
0111111010
4.45
-0.21000004
4.9395995
0.9098712
Exited with code: 0
```

Fig. 3.13: Floating Point Example

3.7 Support for Objective Tiger

I added Objective support to my compiler, which is as explained in the first two chapters.

Chapter 4

Work of Last Semester

During 6^{th} 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.

4.1 What Was 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 4.1 for an example.

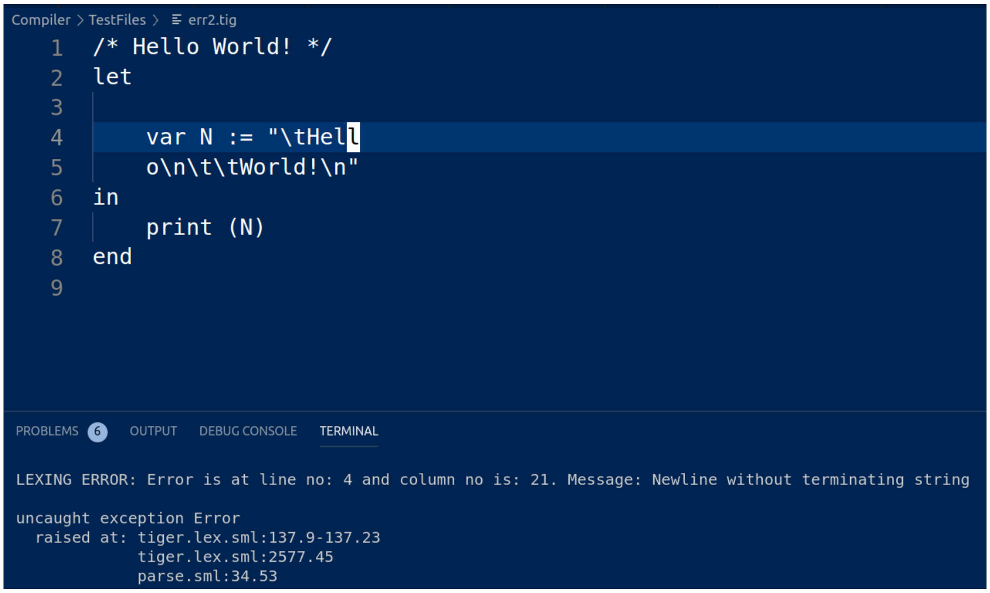


Fig. 4.1: Lexer detecting error where newline is inserted in a string

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 4.2 shows image of the site.

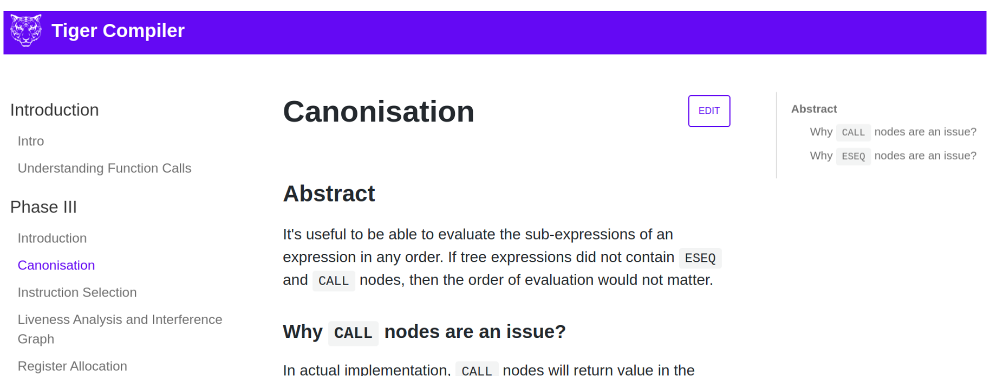


Fig. 4.2: Website: tigercompiler.ml

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 4.3 shows my project at Travis.

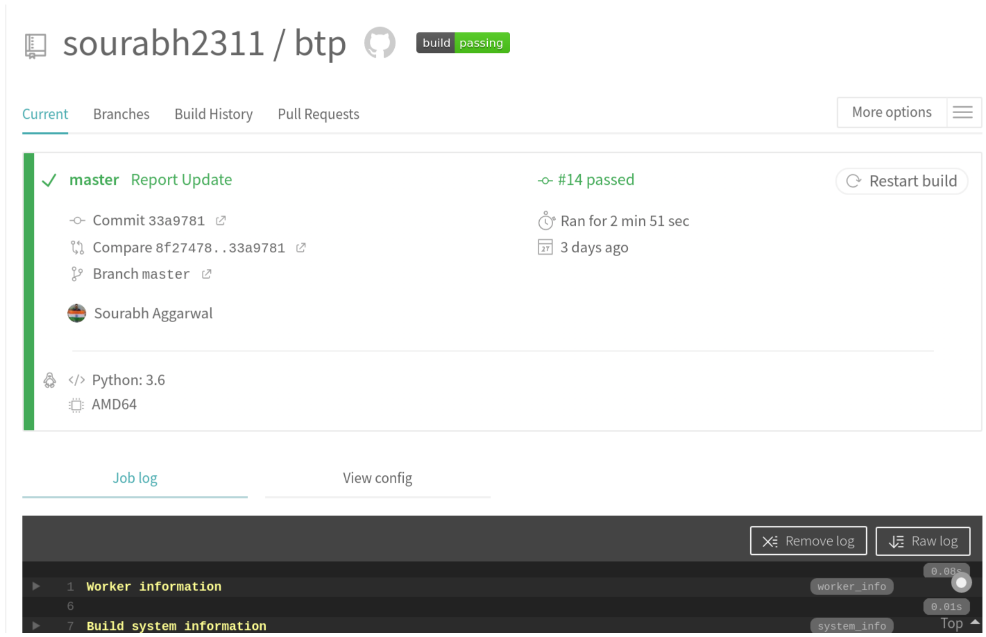


Fig. 4.3: Automated testing using Travis

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 4.5). 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 4.4 shows an example where a lot many arguments are passed to a function and the last argument is returned.

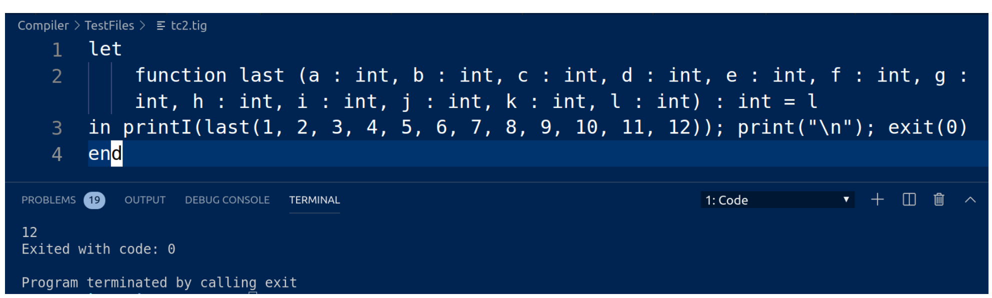


Fig. 4.4: As many function arguments possible

6. Added 2 more arithmetic operations, viz. `left shift` and `right shift`. Figure 4.6 shows an example usage of these operations.

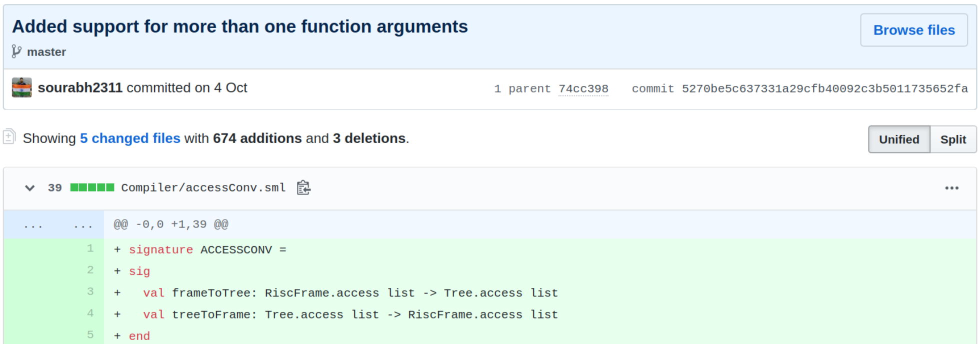


Fig. 4.5: Git commit showing addition of this feature

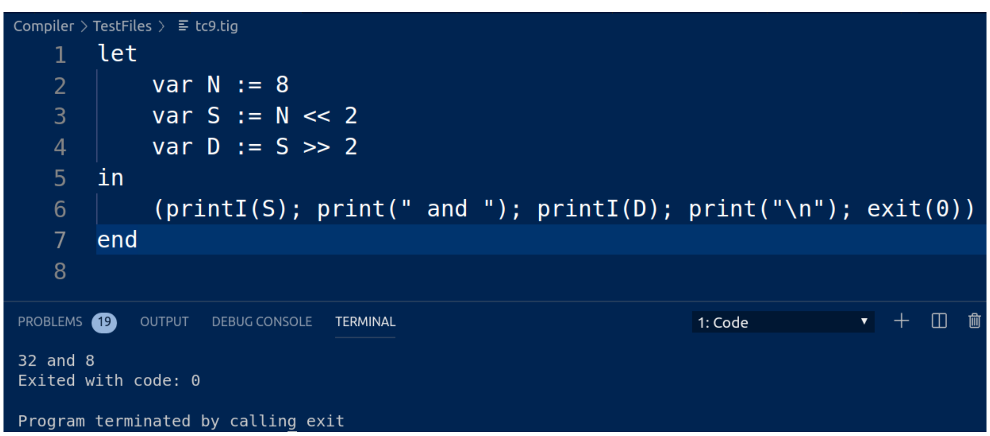
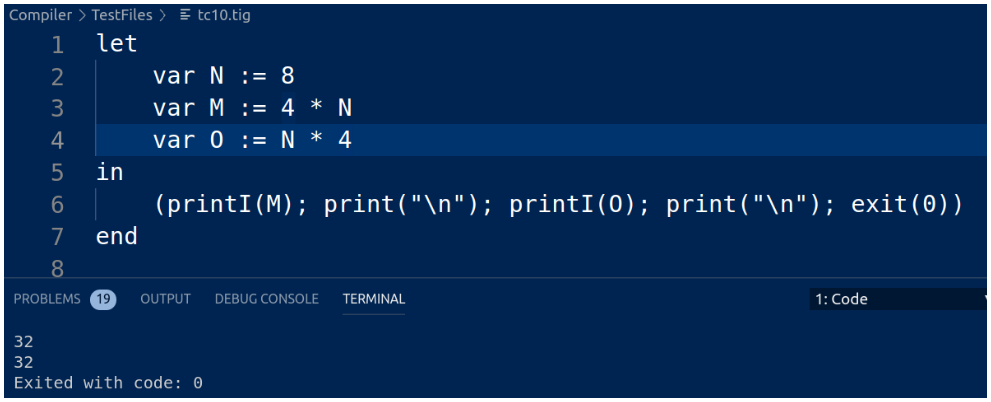
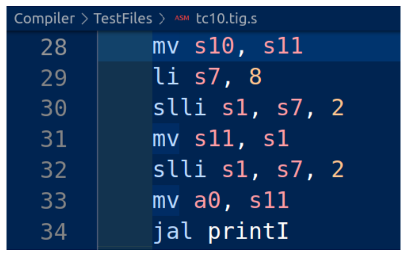


Fig. 4.6: Arithmetic Shift Operations



(a) Code



(b) `mul` statements replaced with `sll`

Fig. 4.7: 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 4.7a and 4.7b.

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.

4.2 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.
- Implement floating point support.
- And much more is possible, its like in a product.

Chapter 5

Introduction to RISC V and Runtime

5.1 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 couldn't find an appropriate documentation & thus after trying various simulators, I chose 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 that 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)

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

5.2 Basic Examples

Few things to keep in mind:

- 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.

5.2.1 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
```

5.2.2 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
```

5.3 Runtime

Associated File(s) Link

runtime.s

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

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 6

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.

6.1 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 → Parsing → ...". `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.

6.2 Intermediate Error Handling

Associated File(s) Link

errormsg.sml

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

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 their been an occurance 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
```

6.3 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.

6.4 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`.

6.5 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.

6.6 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 -> E * E.   (+)
  E -> E. + E   (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`. Also we need not bother by errors generated during parsing, i.e. abstract syntax tree generation phase as ML-YACC will handle it.

Chapter 7

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.

7.1 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.

7.2 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.

7.3 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`

7.4 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.
*)
| RMOVE of exp * exp (* same as above but for reals *)
```

```
  | 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." *)
  | RMEM of exp (* same as above but for reals *)
  | 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
```

7.5 Understanding Functions Calls in Tiger

Note

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

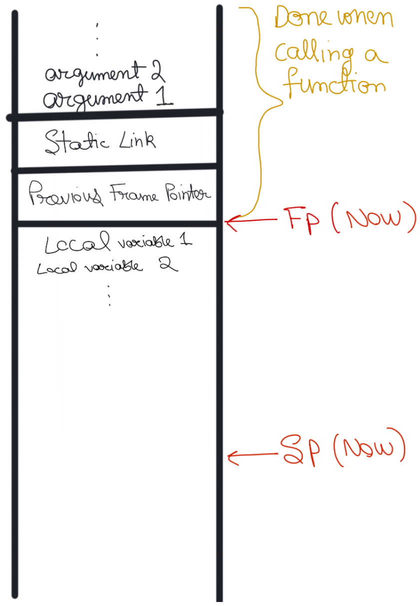


Fig. 7.1: Illustration of Frame

function call is executed in this compiler. Also see 7.1.

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 = [ra] @ 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. For all but those functions which are at top most, we as well pass the static link for the function enclosing the called function. Please see the example code in the text.

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, this frame pointer is not same as static link); 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.

- 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.

7.6 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*[1]. 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.

7.7 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.

`semant.sml` will generate an `exp` of translate type for a frame and then in the end, it will call translate's `procEntryExit` which will `unEx` that expression.

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 | 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.

7.8 Semantic Analysis

Associated File(s) Link

semant.sml

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

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 8

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.

8.1 Canonisation

Associated File(s) Link
canon.sml
Please see comments in the file(s) for more details.

8.1.1 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.

8.1.2 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)`

8.1.3 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.

8.2 Instruction Selection

Associated File(s) Link
assem.sml
codegen.sml
accessConv.sml
Please see comments in the file(s) for more details.

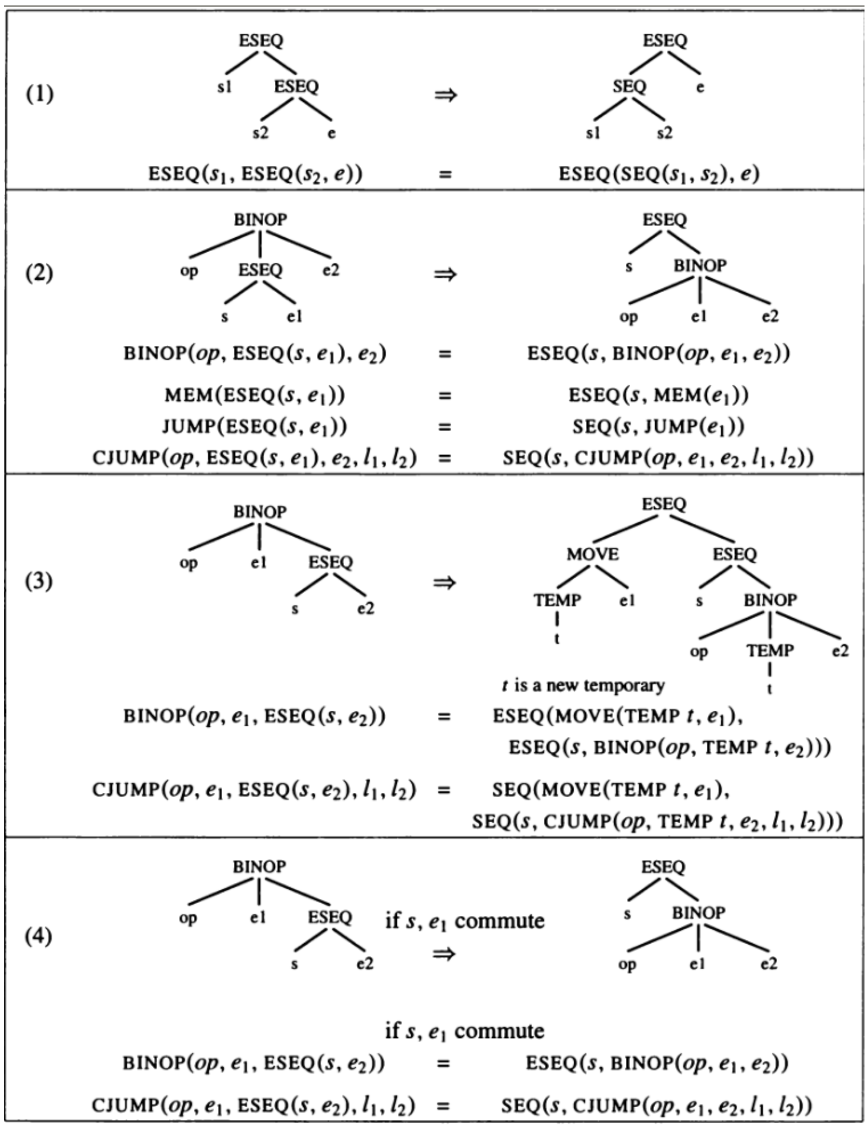


Fig. 8.1: ESEQ Removal. Image source: [1]

8.2.1 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).

8.2.2 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.

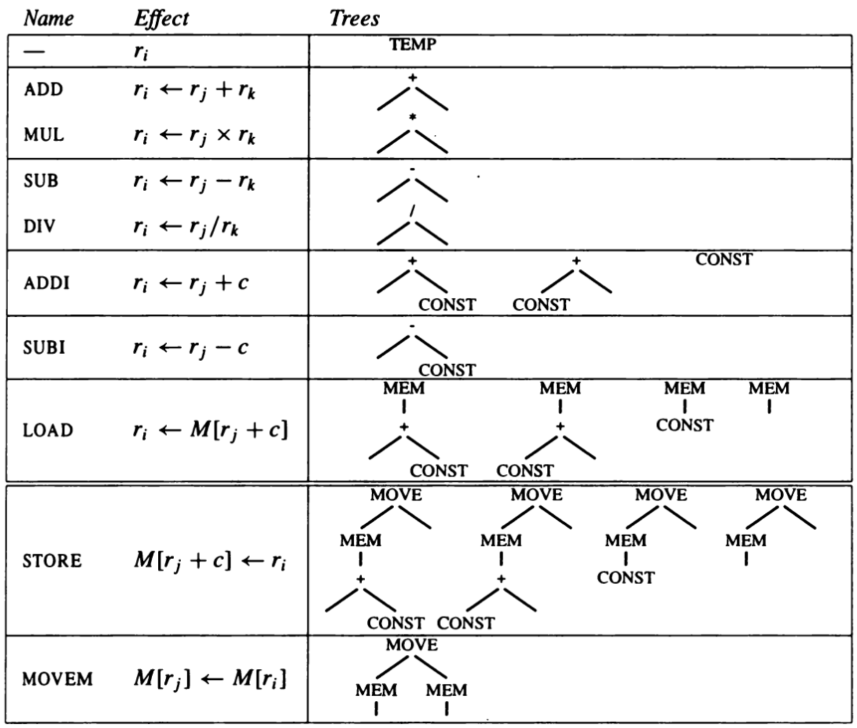


Fig. 8.2: Tree Patterns showing arithmetic and memory instructions. Image source: [1]

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)

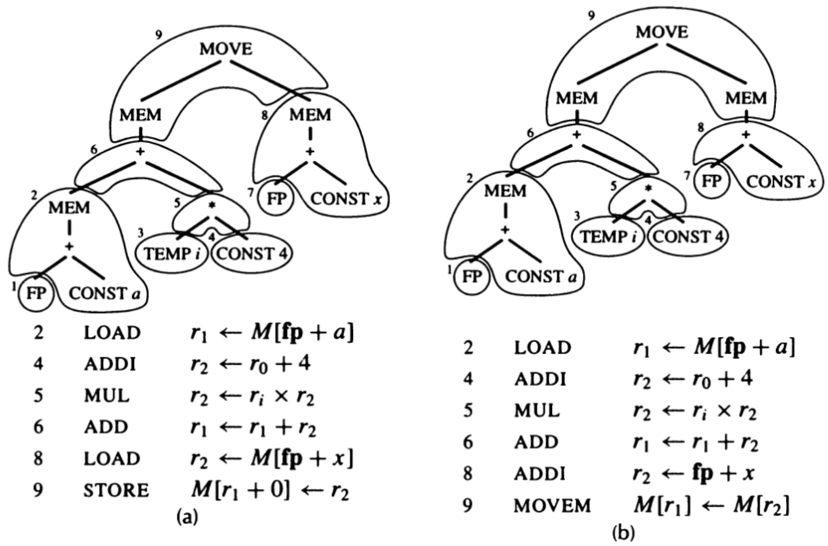


Fig. 8.3: A tree tiled in two ways. Image source: [1]

8.2.3 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`

8.3 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.

8.3.1 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[1]:

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

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

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.

8.3.2 Interference Graph

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

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

8.4 Register Allocation

Associated File(s) Link
regalloc.sml
regalloc.sig
Please see comments in the file(s) for more details.

Here we will be constructing our interference graph and then coloring it. We would follow variety of optimizations and techniques such as Coalescing and Spilling. Since we have two type of temporaries, I have two K's, where K denotes the number of total colors available for a specific temporary. $K1 = |\text{normal saved registers}| + |\text{normal temporary registers}|$, $K2 = |\text{real caller save registers}| + |\text{real callee save registers}|$.

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

8.4.1 Algorithm

The algorithm is as given in the book with few modifications:-

1. Whenever one needs to check for inequality with respect to K then one should call $getK(temporary)$. Where $temporary$ is the concerned temporary.

2. Also when checking for the conservative coalescing, the K to be used will be decided by the temporary which is calling the function for this check.

3. Whenever one is calling $AddEdge$ functions, then before calling it or inside this function, one should check whether both temporaries are real or both are non real. As one cannot add an edge between a real temporary and a non real one as they don't interfere as such.

4. Due to the above modification, one need not store a pair degree map (pair degree map was a degree map where for a temporary it returns a pair denoting number of real and non real neighbours respectively) but can store a simple integer degree map.

5. When assigning colors to a temporary, $okColors$ should be assigned with either $K1$ or $K2$ set of registers depending upon the nature of temporary.

6. On page 248, all the three occurrences of $nodeMoves$ are to be replace by $moveList$. Also add a line below this assignment, $EnableMoves(v)$.

7. AND most importantly, there is a major bug in their algorithm, which is in procedure $FreezeMoves(u)$, where in the last if condition, they should also add a check that whether v is not precolored because we cannot simplify a precolored nodes.

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