Manual of FDMJ Compiler

Author: **Qijun Feng**

Supervisor: Xiaoyang Wang

Spring 2024

Contents

1 Introduction

This report serves as a documentation and manual book of the FudanCompiler, which compiles a custom language named Fudan Mini Java (FDMJ) into either LLVM or arm instructions. Our target readers are computer science students without learning compiler courses.

The compiler is designed to perform the following key stages of compilation: lexical analysis and parsing (Section 2), type checking (Section 3), translation to IR (Section 4), canonicalization (Section 5), LLVM instruction selection (Section 6), static single assignment (Section 7), RPi instruction selection (Section 8), and register allocation (Section 10). Finally, we will briefly introduce the file structure and usage of our compiler in Section 11.

The grammar of FDMJ language and some supplementary explanation is detailed in Appendix A. Throughout the report, the following program will be used as a simple yes illustrative example:

```
public int main() {
    int[] a={0};
    int i=0;
    int l;
    class c1 o1;
    class c1 o2;

    a=new int[getnum()];
    l=length(a);
    o1=new c1();
    o2=new c2();

    while (i < l) {
        if ((i/2)*2 == i)
            a[i]=o1.m1(i);
        else
            a[i]=o2.m1(i);
        i=i+1;
    }
    putarray(l, a);
    return l;
}
```

```
public class c1 {
    int i1=2;
    public int m1(int x) {
      return this.i1;
    }
}

public class c2 extends c1 {
    // int i1=3;
    public int m1(int x) {
        return this.i1+x;
    }
}
```

2 Lexical Analysis and Parsing

Lexical Analysis is aimed to break the input into individual words or "tokens". I quote from **Modern Compiler Implementation in C**: The lexical analyzer takes a stream of characters and produces a stream of names, keywords, and punctuation marks; it discards white space and comments between the tokens.

The general approach is to specify lexical tokens using the formal language of regular expressions. Since regular expressions are equivalent to DFAs, a lexical analyzer can be implemented using a deterministic finite automaton (DFA). However, constructing a DFA is a tedious task that can be easily handled by a computer. Therefore, we can use a lexical analyzer generator like `Lex` to convert regular expressions into a DFA.

Our `Lex` file records the position of keywords and return a symbol in the `rule` part. Moreover, our lexer is able to handle two kinds of comments, *i.e.* `/* */` and `//`. Here is a piece of example:

```
<INITIAL>"float" {
  yylval.pos = A_Pos(line, pos);
  pos += yyleng;
  return FLOAT;
}
<INITIAL>"getnum" {
  yylval.pos = A_Pos(line, pos);
  pos += yyleng;
  return GETNUM;
}
```

After the lexical analysis, we further parse the source code into a tree structure (AST). Context-Free Grammar (CFG) specifies the syntax rules, and using a parser generator like Yacc to convert these rules into a parsing table and execute the corresponding syntax actions is a convenient approach.

Yacc primarily targets LR(1) grammars. By constructing LAIR(1) tables and declaring precedence to resolve conflicts, it can handle not only LR(1) grammars but also ambiguous grammars.

Practically, based on the syntax of FDMJ, we define terminal symbols, non-terminal symbols, and the start symbol. We specify the grammar in this way:

```
ARDECLLIST: /* empty */ {
  $$ = NULL;
} | VARDECL VARDECLLIST {
  $$ = A_VarDeclList($1, $2);
} ;
STM: '{' STMLIST '}' {
  $$ = A_NestedStm($1, $2);
} | IF '(' EXP ')' STM  ELSE STM {
  $$ = A_IfStm($1, $3, $5, $7);
} | IF '(' EXP ')' STM %prec IF{
  $$ = A_IfStm($1, $3, $5, NULL);
} | WHILE '(' EXP ')' STM {
  $$ = A_WhileStm($1, $3, $5);
} | WHILE '(' EXP ')' ';' {
  $$ = A_WhileStm($1, $3, NULL);
} | EXP '=' EXP ';' {
  $$ = A_AssignStm(A_Pos($1->pos->line, $1->pos->pos), $1, $3);
} | EXP '[' ']' '=' '{' EXPLIST '}' ';' {
  $$ = A_ArrayInit(A_Pos($1->pos->line, $1->pos->pos), $1, $6);
} | EXP '.' ID '(' EXPLIST ')' ';' {
  $$ = A_CallStm(A_Pos($1->pos->line, $1->pos->pos), $1, $3->u.v, $5);
} | CONTINUE ';' {
  $$ = A_Continue($1);
} | BREAK ';' {
  $$ = A_Break($1);
} | RETURN EXP ';' {
  $$ = A_Return($1, $2);
  ...
```

Here the `A_XXX`s are carefully designed abstract syntax tree (AST) data structures, and the `$NUM` in arguments indicate the corresponding number of non-terminal in the grammar above. See `fdmjast.h|c` for more details. By setting precedence to eliminate shift-reduce conflicts, we ultimately obtain the abstract syntax tree. Given that the complete AST is too long to present, we only give parsing result (in XML format) of `a = new int[getnum()];` for reference:

```
<assign>
<left>
<id>
<pos><line>8</line><col>5</col></pos>
<string>a</string></id>
</left>
<right>
<newInt>
<pos><line>8</line><col>7</col></pos>
<getnum>
<pos><line>8</line><col>15</col></pos>
</getnum>
</newInt>
</right>
</assign>
```

3 Type Checking

After parsing, the abstract syntax tree (AST) cannot be translated into LLVM IR instructions directly since not all syntactically correct programs are semantically valid. Therefore, we need to perform type checking.

Type checking mainly involves checking the classes and the main method. We use the following auxiliary data structures to assist with type checking:

- `cenv`: an S_table to record classes. To be specific, each class has an entry consists of its name, its father, a variable table `vtbl`, and a method table `mtbl`.
- `vtbl`: an S_table that records mappings from a class variable to its declaration, type and `Temp_temp`.
- `mtbl`: an S_table that records mappings from a class method to its declaration, class, return type and formal list.
- `venv`: an S_table to record variables

The check of classes is divided into three passes:

- First pass: record the name of each class and its father (if any). Based on the recorded inheritance relationships, determine if there is any circular inheritance.
- Second pass: record the class variables and signatures (including return type and parameter types) of methods in each class. If a class has a father, process its father first, copying its variable table and method table, then check the current class. This pass helps to detect the redefining of variables and method. Note that a method can be "redefined" only if its signature is exactly the same. During this pass the `vtbl` and `mtbl` of each class is constructed and stored in `cenv`.
- Third pass: enter each class method for detailed checked (discussed below).

Each time we enter a method, we call `S_beginScope(venv)` to create an empty variable environment. Similarly, `S_endScope(venv)` is called when leaving a method. The detailed checklist within each method and the corresponding error message is provided below:

- check any variable redefined; if it is the case, report *Variable re-declaration.*
- check the condition of `if` and `while` statement is `int` or `float`; if it is not the case, report *Incompatible if/while statement condition.*
- check `break` and `continue` are in `while` loop; if it is not the case, report *Break/-Continue must be in while loop.*
- for assign statement, check whether the type of both sides is compatible (note that upcasting is allowed in FDMJ, *i.e.* `Father o = new Son()`), and whether the left operator has a location; however, implicit casting between an `int array` and a `float array` is not allowed; if it is not the case, report *Incompatible type* or *Must assigned to a left value.*
- check whether both sides of an operator are `int` or `float`; if it is not the case, report *Incompatible operand type! (int/float expected).*
- check whether a class, class method, class variable exists when called; if it is not the case, report *Undefined class method/variable* or *Not a class type.*
- check whether the arguments of a method is correct (in terms of number and type); if it is not the case, report *Incompatible argument type! (int/float expected)* or *Incompatible parameter number.*

- check whether an array is initialized properly; if it is not the case, report *Incompatible type*.
- check whether `this` is within a class method; if it is not the case, report *This expression in main method*.

The error message is presented in the terminal by `transError` method below, which will also present the position where the error occurred. Now the necessity of recording the position of the keywords is demonstrated, in this way, we are able to provide users with a clearer direction to fix the errors.

```
void transError(FILE *out, A_pos pos, string msg) {
    fprintf(out, "(line:%d col:%d) %s\n", pos->line, pos->pos, msg);
    fflush(out);
    exit(1);
}
```

4 Translation to IR+

TigerIR+ is an intermediate representation that serves as a ladder of the following low level instruction selection but preserves the preferred tree structure. By translating the abstract syntax tree (AST) into IR+, the concept of class is "eliminated", only a bunch of expressions are left.

We adopt the design of performing type checking and translation simultaneously, which is efficient and tidy. To enable isolation of these two procedures, type checking is done in the body of `transA_XXX` functions (found in `semant.h|c`), while translation is handled in the return part of `transA_XXX` functions by calling the corresponding translation functions `Tr_XXX` (found in `translate.h|c`).

Since type checking has already been specified in Section 3, here we mainly focus on the details of translating. We first introduce the concept and implementation of **unified object record**.

Unified object record is a simplified way to handle object (*i.e.* classes), which leaves space for all possible variables and methods in all classes. In this way, each name (variable or method) corresponds to one unique offset in the record. We initialize and call (use) the class variables and methods by checking their offset relative to the begin address of the object. Meanwhile, the inheritance is handled by linking the class name and method name with a `$` during compile time. The unified object record for the example in Figure 4.1 is `(x,y,f,g)`. And we allocate $4 \times arch\_size$ space for any object.

```
class A extends Object {
                    var x := 0
                    method f() }
class B extends A {method g() }
class C extends B {method g() }
class D extends C {var y := 0
                    method f() }
```

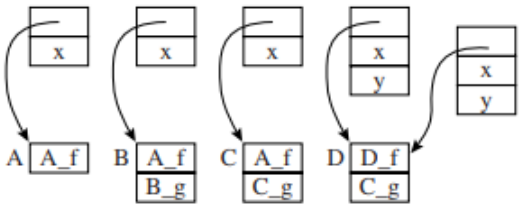

Figure 4.1: An example of unified object record

Using the example in Section 1, the unified object record is `(i1,m1)`, which requires $2 \times arch\_size$ space for any object. Note that here arch_size is set to 8. So the statement `o1 = new c1()` is translated to

```
T_Move(
    T_Temp(Temp_namedtemp(106,T_int))/*T_int*/,
    T_Eseq(
      T_Seq(
        T_Seq(
          T_Move(
            T_Temp(Temp_namedtemp(110,T_int))/*T_int*/,
            T_ExtCall(String("malloc"),
              T_ExpList(
                T_IntConst(16)/*T_int*/,
                NULL)
            ,T_int)/*T_int*/
          ),
          T_Move(
            T_Mem(
              T_Binop(T_plus,
                T_Temp(Temp_namedtemp(110,T_int))/*T_int*/,
                T_IntConst(8)/*T_int*/
              )/*T_int*/
            , T_int)/*T_int*/,
            T_Name(Temp_namedlabel(String("c1$m1")))/*T_int*/
          )
        ),
        T_Move(
          T_Mem(
            T_Binop(T_plus,
              T_Temp(Temp_namedtemp(110,T_int))/*T_int*/,
              T_IntConst(0)/*T_int*/
```

```
          )/*T_int*/
        , T_int)/*T_int*/,
        T_IntConst(2)/*T_int*/
      )
    ),
    T_Temp(Temp_namedtemp(110,T_int))/*T_int*/
  )/*T_int*/
),
```

Again several auxiliary data structures are leveraged in translation.

- `test_stack` and `end_stack`: store the test label and end label of a `while` loop, respectively.
- `var_offset`: record the offset of a class variable.
- `meth_offset`: record the offset of a class method.

Finally, we discuss how various components in AST is translated:

- `T_Binop` is used to handle arithmetic operators, joining both operands together. It returns `Tr_Ex`.
- Comparison operators is treated as a condition. We leave the true label and false label for later filling and it returns `Tr_Cx`.
- It is worth noting that in FDMJ, the logical operators `&&` and || in conditional statements have shortcut logic during translation, making them relatively complex. Specifically,
  - For `&&` operator, we fill the false label of the first expression with the next label, patch the true labels of both expressions for later filling, and also leave the false label of the second expression for later filling.
  - For || operator, we fill the true label of the first expression with the next label, patch the false labels of both expressions for later filling, and also leave the true label of the second expression for later filling.
- `if` statement is handled as follows:
  - If there is no `else`, when the condition of the `if` statement is true, jump to the `then` label, and return to the main program. If the condition is false, return to the main program.

  - If there is an `else`, when the condition of the `if` statement is true, jump to the `then` label, and return to the main program. If the condition is false, jump to the `else` label, and return to the main program.

- To handle `while` statement, we utilize the aforementioned stacks to store the corresponding test label and end label. If there is a loop, when test condition is true, jump to the loop label, and return to the test label; otherwise, jump to the end label. When encountering a `break`, we can directly pop the end label stack and jump to the end label. When encountering a `continue`, we can similarly pop the test stack and jump to the test label.

- The initialization and assignment of an array works as follows: we `malloc` a continuous space of size $(length + 1) \times arch\_size$ and plus the returned address with arch_size as the beginning address of the array. Then we access each element of the array with its offset.

- `length` of an array is stored at the -1 position of the array when initialized, so we can just load it like normal array expression.

5 Canonicalization

The trees generated by the semantic analysis phase must be translated into assembly or machine language. However, there are certain aspects of the tree language that do not correspond exactly with machine languages, and some aspects of the `Tree` language interfere with compile-time optimization analyses.

The mismatches between `Tree`s and machine-language programs are:

- `CJUMP` instructions that can jump to either of two labels, but real machine conditional instructions fall through to the next instruction if the condition is false.

- `ESEQ` nodes with expressions make different orders of evaluating subtrees yielding different results.

- `CALL` nodes within expressions causes the same problem.

- `CALL` nodes within the argument-expression of other `CALL` nodes will cause problems when trying to put arguments into a fixed set of formal-parameter registers.

In this way, we can rewrite the tree into equivalent tree without any of the cases listed above. Without these cases, the only possible parent of a `SEQ` node is another `SEQ`, so we might as well get rid of them and make a linear list of `T_stm`s.

This 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; finally the basic blocks are ordered into a set of traces in which every `CJUMP` is immediately followed by its false label.

Practically, we just follow the aforementioned strategy and do linearization, chunk the `Stm`s into basic blocks and call `traceSchedule` to reorder the basic blocks. See `canon.h|c` for more implementation details. The example in Section 4 is canonicalized to

```
Move t110:int, malloc 16:int:int
Move Mem(Binop(T_plus, t110:int, 8:int):int):int, c1$m1:int
Move Mem(Binop(T_plus, t110:int, 0:int):int):int, 2:int
Move t106:int, t110:int
```

6 LLVM Instruction Selection

After obtaining IR and performing canonicalization, which marks the end of front end, all classes "disappear". In this part, we are just dealing with statements. Intuitively, what LLVM instruction selection does is tiling from top to bottom and then emitting LLVM IR instructions from bottom to top.

Since there is much less cases than previous parts, we detail the tiling strategy of each case below. For `T_stm`,

- `T_SEQ`: munch both sides respectively.
- `T_LABEL`: emit the instruction with the label.
- `T_JUMP`: emit the `br label` instruction.
- `T_CJUMP`: it corresponds to two consecutive instructions where the first compute the result of comparison and store it in a one-bit variable while the second tells which label to jump to based on the one-bit bool condition.

  ```
  %L = icmp CND i64 OP1, OP2 / %L = fcmp CND i64 OP1, OP2
  br i1 <cond>, label <iftrue>, label <iffalse>
  ```

- `T_EXP`: calls the `llvmMunchExp` to tile.
- `T_RETURN`: emit `ret i64` or `ret double` according to the return temporary type.

For `T_Exp`,

- `T_BINOP`: emit instructions based on the operator and operands' type.
- `T_MEM`: we first convert the source address into a pointer, and load data from it:

```
%%`d0 = inttoptr i64 %%`s0 to i64*
%%`d0 = load i64, i64* %%`s0
```

- `T_TEMP`: return the corresponding `Temp_temp` directly.
- `T_CONST`: make a new `Temp_temp`, and emit an instruction `%L = add i64/double CONST, 0`. Return the new temporary.
- `T_NAME`: make a new `Temp_temp`, and emit an instruction `%L = ptrtoint i64* @%s to i64`. Return the new temporary.
- `T_ESEQ`: call the `T_Stm`s first and return the result of the `T_Exp`.

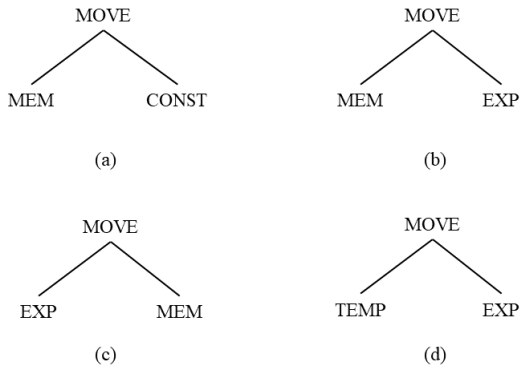

Figure 6.1: Tree of T_MOVE.

`T_MOVE` nodes are relatively complex. As illustrated in Figure 6.1, we can categorize them into roughly four cases.

- (a): first use `inttoptr` to transform the `i64` address into `i64*`, then `store` the constant to that address.
- (b): first use `inttoptr` to transform the `i64` address into `i64*`, then munch the EXP, and finally `store` the temporary representing the EXP to that address.
- (c): first use `inttoptr` to transform the `i64` address into `i64*`, then `load` the int/double from that address.
- (d): this is just regular "assign" operation.

See `llvmgen.h|c` for more implementation details. Following the previous example, the LLVM instructions correspond to the canonical IR are:

```
%r133 = call i64* @malloc(i64 16)
%r110 = ptrtoint i64* %r133 to i64
%r134 = ptrtoint i64* @c1$m1 to i64
%r135 = add i64 %r110, 8
%r136 = inttoptr i64 %r135 to i64*
store i64 %r134, i64* %r136
%r137 = add i64 %r110, 0
%r138 = inttoptr i64 %r137 to i64*
store i64 2, i64* %r138
%r106 = add i64 %r110, 0
```

7 Static Single Assignment(SSA)

One of the most important optimizations on LLVM instructions is static single assignment, namely, each variable will only be assigned once with the assistance of phi function after this process.

The static single assignment can be divided into several steps: compute dominance relationship based on liveness analysis, compute dominance frontier of each node, place phi functions in proper positions, and finally rename the variables. Liveness analysis will be specified in Section 9 and here we assume the liveness graph and basic block graph are obtained. Before renaming variables, all operations are performed on a block basis.

Preparation

To facilitate the later procedures, several preprocessings are required. First, we iterate through the liveness graph and basic block graph to record the defsites of each variable. Then, we iterate the liveness graph again to record if a variable has been defined more than once. In this part, many auxiliary tables are utilized to record information. The keys are all block labels and the values varies from dominators, immediate dominator, original definition places, instruction lists, and dominance frontier.

Compute dominance relations

We need to compute the dominators of each node according to the following equation.

$$D[s_0] = \{s_0\} \quad D[n] = \{n\} \cup \left(\bigcap_{p \in pred[n]} D[p] \right) \tag{7.1}$$

The implementation is straightforward except that $D[n]$ should be initialized to all nodes in the graph since the intersection operation will reduce elements.

After that, we can readily compute the immediate dominator of each node (if exists) by traversing the basic block graph again.

Compute dominance frontier

According to the equation below, the dominance frontier of node n is:

$$DF[n] = DF_{local}[n] \cup \bigcup_{c \in children[n]} DF_{up}[n] \tag{7.2}$$

We can just implement Algorithm 1 to obtain the dominance frontier of each node.

Algorithm 1: Compute dominance frontier

```
Data: The starting block n
Result: The computed dominace frontier of all blocks in the given graph
S ← {} ;
for each node y in succ[n] do
    if idom(y) ≠ n then
        S ← S ∪ {y} ;
    end
end
for each child c of n in the dominator tree do
    computeDF[n] ;
    for each element w of DF[c] do
        if n does not dominate w then
            S ← S ∪ {w} ;
        end
    end
end
DF[n] ← S ;
```

Place phi functions

Starting with a program not in SSA form, we need to insert just enough ϕ-functions to satisfy the iterated dominance frontier criterion. The worklist W contains nodes that may violate the dominance frontier criterion.

Practically, we further improve Algorithm 2 by modifying the "if Y $\notin$ DF[n]" to "if Y $\notin$ DF[n] and a live-out in Y". In this way, we can eliminate some unnecessary insertion of ϕ-functions.

Rename variables

Algorithm 3 is a rough skeleton for reference. I make several adjustments so that it works better with our LLVM instructions. For those variables with only one definition, we do not change their name. A stack is kept for the rest variables (in the form of `Temp_temp`). Each time we encounter a definition of such variable, we create a new `Temp_temp` and

Algorithm 2: Inserting ϕ-functions

Result: The instruction list after inserting ϕ-functions in proper places

for *each node n* **do**
  for *each variable a in* $A_{orig}[n]$ **do**
    defsites[a] $\leftarrow$ defsites[a] $\cup$ {n} ;
  end
end
for *each variable a* **do**
  W $\leftarrow$ defsites[a] ;
  while *W not empty* **do**
    remove some node n from W ;
    for *each Y in DF[n]* **do**
      if $Y \notin A_\phi[n]$ **then**
        insert the statement a $\leftarrow \phi(a, \ldots, a)$ at the top of block Y ;
        $A_\phi[n] \leftarrow A_\phi[n] \cup \{Y\}$;
        if $a \notin A_{orig}[Y]$ **then**
          W $\leftarrow$ W $\cup$ {Y}
        end
      end
    end
  end
end

replace all the uses of this variable along its dominance (including phi functions). When all its child in the dominator tree has been visited, we pop this `Temp_temp` from the stack. In other words, at any timestep, a variable has one latest "version" and we ensure every use along its dominance of this variable is replaced.

Eliminate phi functions

At this point, the instructions can run with `lli` since `LLVM` supports ϕ-functions (refer to Section 11 for details). Yet there is no ϕ-functions in `ARM` instruction set. Hence we will have to "eliminate" the ϕ-functions by converting them into several `mov` instructions. For each ϕ-function, we find the predecessors of the block and add a `mov` instruction just before the `br` instruction, which has exactly the same semantic (behavior) of ϕ-functions.

8 RPi (ARM) Instruction Selection

Instead of converting IR+ to ARM instructions directly, we opt for translating SSA form of LLVM instructions to ARM instructions.

Since different kinds of LLVM instructions (may) require different converting strategy, we first decide the type of instruction by string matching:

```
if (!strncmp(assem, "br label", 6)) {
    ret = BR;
```

Algorithm 3: Renaming variables

```
Initialization ;
for each variable a do
    Count[a] ← 0 ;
    Stack[a] ← empty ;
    push 0 onto Stack[a] ;
end
Rename(n) ;
for each statement S in block n do
    if S is not a φ-function then
        for each use of some variable x in S do
            i ← top(Stack[x]) ;
            replace the use of x with x_i in S ;
        end
    end
    for each definition of some variable a in S do
        Count[a] ← Count[a] + 1 ;
        i ← Count[a] ;
        push i onto Stack[a] ;
        replace definition of a with definition of a_i in S ;
    end
end
for each successor Y of block n do
    Suppose n is the jth predecessor of Y ;
    for each φ-function in Y do
        suppose the jth operand of the φ-function is a ;
        i ← top(Stack[a]) ;
        replace the jth operand with a_i
    end
end
for each child X of n do
    Rename(X) ;
end
for each definition of some variable a in the original S do
    pop Stack[a] ;
end
```

```
    return ret;
} else if (!strncmp(assem, "ret", 3)) {
    ret = RET;
    return ret;
} else if (!strncmp(assem, "%`d0 = fadd", TYPELEN)) {
    ret = FADD;
    return ret;
    ...
```

Since there are several mismatches between LLVM instructions and ARM instructions, we translate them based on the type of instruction:

- For binary operations, we just `mov` the constants (if there is) into temporaries and only need to handle binary operations between temporaries.
- `P2I` and `I2P` basically refers to a simple `mov` except for instruction involving method name (*e.g.* `@c1$m1`), we use `ldr %%`d0, =%s` to handle this.
- For `LOAD`, we use `ldr d0, [s0]` to translate, where `d0` and `s0` corresponds to the source address and destination temporary, respectively.
- For `STORE`, we use `str s0, [s1]` to translate.
- For `CMP`, we use `cmp s0, s1` to compare two temporaries and record the comparison operator (*e.g.* eq, ne). The next instruction must be a `BR` one which requires to operator to emit an instruction `beq j0`.
- If the temporaries to compare are two floating points, we will have to add a `vmrs APSR_nzcv, FPSCR` after the comparison.
- What makes things much more complicated are `CALL`s. Until this point, we have been only handling temporaries. But since the arguments in ARM calls are passed by registers `r0 - r3`, we will have to move the corresponding temporaries into these arm registers before calling any function. Moreover, registers `r0 - r3` are caller saved registers, so each time we make a call, we will have to store them in stack. Meanwhile, registers `r4-r16` are callee saved registers, namely, we must store them in stack whenever we are entering a function.

Note that integers and floating points must be handled by correct instructions, *e.g.*, we should use `vadd.f32` instead of `add` to add two floating points.

9 Liveness Analysis

The front end of the compiler translates programs into an intermediate language with an unbounded number of temporaries. Yet this program must run on a machine with a bounded number of registers. A direct motivation of doing liveness analysis is that we want to figure out whether two temporaries a and b can fit into the same register. If a and b are never "in use" at the same time, then they can be allocated to the same register.

Consider the program and its flow graph in Figure 9.1. From each statement, we can easily tell its *def* and *use*. Here $def(3) = \{c\}$, $use(3) = \{b,c\}$.

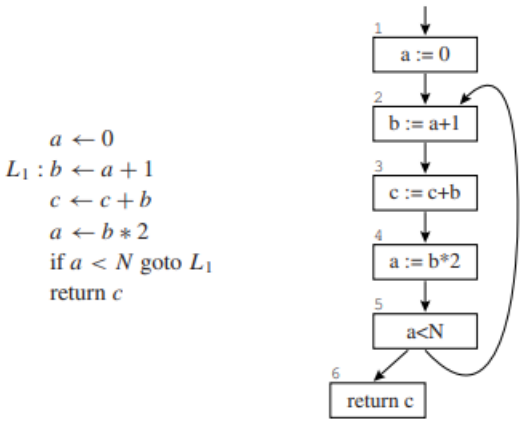

Figure 9.1: An example program.

In our assembly instructions (including LLVM and RPi), the *def* is the *dst* list and *use* is the *src* list. Based on the dataflow equations below, we can derive an iterative way of computing liveness.

$$in[n] = use[n] \cup (out[n] - def[n]) \tag{9.1}$$

$$out[n] = \bigcup_{s \in succ[n]} in[s] \tag{9.2}$$

We follow the convention that calculate liveness in reverse control flow edge order. We initialize the *in* and *out* set of each node with an empty set and use the above equation to update them until convergence. An example of this process is illustrated in Figure 9.2. We also provide the liveness analysis result in Appendix A to further show how this mechanism works in our compiler.

One of the most useful application of liveness analysis is for register allocation. A condition that prevents two temporaries a and b to be allocated to the same register is called an *interference*. The most common kind of interference is caused by overlapping live ranges.

			1st		2nd		3rd	
	use	*def*	*out*	*in*	*out*	*in*	*out*	*in*
6	c			c		c		c
5	a		c	ac	ac	ac	ac	ac
4	b	a	ac	bc	ac	bc	ac	bc
3	bc	c	bc	bc	bc	bc	bc	bc
2	a	b	bc	ac	bc	ac	bc	ac
1		a	ac	c	ac	c	ac	c

Figure 9.2: Liveness calculation following reverse control flow edges.

We derive and implement two criteria to add interference edges for each new definition:

- At any non-move instruction that defines a variable a, where the live-out variables are $b_1, \ldots, b_j$, add interference edges $(a, b_1), \ldots, (a, b_j)$.
- 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.

And here is the implementation.

```
for (Temp_tempList tl1 = FG_def(n); tl1 != NULL; tl1 = tl1->tail) {
  Look_ig(tl1->head);
  for (Temp_tempList tl2 = FG_Out(n); tl2 != NULL; tl2 = tl2->tail) {
    if ((!(is_m && use != NULL && tl2->head == use->head))
        && (tl2->head != tl1->head)) {
      Enter_ig(tl1->head, tl2->head);
    }
  }
}
```

The interference graph of the example given in Section 1 can be found in Appendix A.

10 Register Allocation

In the previous sections, we made a critical assumption (simplification) that there is numerous temporaries (registers). However, no matter which system we use, the number of available registers is always limited. So we have to allocate a register for every temporary so that the program can run correctly and it should be semantically consistent with the one before doing register allocation.

First, we construct an interference graph based on the ARM instructions, where each node represents a variable. An edge between two nodes indicates that these two vari-

ables cannot be allocated to the same register. Thus, the register allocation problem is equivalent to a graph coloring problem.

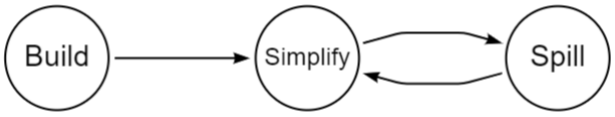

Figure 10.1: The overall pipeline of register allocation.

The degree of each node indicates the number of temporaries that interferes with it, *i.e.* they cannot be assigned to the same register. A conservative way of doing register allocation is iteratively find the temporaries whose degree is less than the number of available registers, and delete it (and its edges) from the graph. This process is termed as "simplify". Practically, we keep the simplified node in a stack for later use. If we are lucky enough, all nodes in the graph can be simplified in this way. However, if all nodes in the graph can no longer be simplified, namely, all nodes have degree greater than or equal to the number of available registers, we "spill" the node with the greatest degree. In practice, we just mark it as spilled and delete it from the graph. Then we return to the "simplify" step and stop this process until the graph is empty.

Finally, we pop the nodes from the aforementioned stack and allocate them a color (*i.e.* a register number). Each time we choose a color different from all its neighbors, and given the previous simplification process, we can always find a color for the nodes in the stack. For those spilled temporaries, we assign and book-keep an offset for them, and do `store` and `load` for each use and def.

11 File Structure and Usage

The `include/` directory contains all the headers of the compiler and their implementation is in the `lib/` directory. Both directories have the same file structure. The test files are in the `test/` directory.

The whole compiler can be compiled with a simple `make compile` command. And we provide the following commands for convenient usage:

- `make run-llvm`: compile all the file under the test directory into LLVM instructions and run them with `lli`.
- `make run-arm`: compile all the file under the test directory into ARM instructions and run them on `QEMU`.

See `README` under the root directory for more details.

12 Conclusions

It is really an amazing experience to implement a compiler from scratch during a semester. I have to say that debugging a compiler is extremely hard–there must have been a thousand times that I want to quit. But I persevered!! Seeing the program written in FDMJ gradually converts from AST, intermediate representation, LLVM instructions, to the final ARM instructions, I feel that all the works paid.

The difference between theories in the book and the problems encountered when you actually build a compiler. We must not only fully understand the big picture of each stage, but also pay careful attention to details. Dealing with corner cases properly is by no means some dispensable thing as a compiler is supposed to handle strange input that might go beyond the consideration during designing.

Thanks again to Professor Xiaoyang Wang, and TAs Jiangfan and Yanjun. It would be impossible for me to complete the project without your detailed documents, auxiliary programs and every-week's Q&A time.

A Appendix

FDMJ Grammar

This is the mini programming language we are going to compile in this project.

```
Program -> MainMethod ClassDecl*
MainMethod -> public int main '(' ')' '{' VarDecl* Statement* '}'

ClassDecl -> class id [extends id] '{' VarDecl* MethodDecl* '}'

Type -> class id | int | int '[' ']' | float | float '[' ']'
VarDecl -> Type id ';'   Type id '=' CONST ';' |
            Type id '=' '{' ConstList '}' ';'

ConstList -> CONST ConstRest* | \empty
    ConstRest -> ',' CONST

MethodDecl -> public Type id '(' FormalList ')' '{' VarDecl* Statement* '}'

FormalList -> Type id FormalRest* | \empty
    FormalRest -> ',' Type id

Statement ->
    '{' Statement* '}' |
    if '(' Exp ')' Statement else Statement |
    if '(' Exp ')' Statement |
    while '(' Exp ')' Statement |
    while '(' Exp ')' ';' |
    id = Exp ';' |
    id '[' Exp ']' = Exp ';' |
    id '['  ']' = '{' ExpList '}' ';' |
    Exp '.' id '(' ExpList ')' ';' |
    continue ';' | break ';' |
    return Exp ';' |
    putint '(' Exp ')' ';' | putch '(' Exp ')' ';' |
    putint '(' Exp ')' ';' | putch '(' Exp ')' ';' |
    putarray '(' Exp ',' Exp ')' ';' |
    starttime '(' ')' ';' | stoptime '(' ')' ';'

Exp -> Exp op Exp |
```

```
    Exp '[' Exp ']' |
    Exp '.' id '(' ExpList ')' |
    Exp '.' id |
    CONST |
    true | false |
    id | this | new int '[' Exp ']' | new float '[' Exp ']' |
    new id '(' ')' | '!' Exp | '-' Exp | '(' Exp ')' |
    '(' '{' Statement* '}' Exp ')' |   //escape expression
    getint '(' ')' | getch '(' ')' | getarray '(' Exp ')'

ExpList -> Exp ExpRest* | \empty
    ExpRest -> ',' Exp
```

The semantic of an FDMJ program with the above grammar is similar to that for programming language of C and Java. Here we give a few notes about it:

- The root of the grammar is `Program`.
- The binary operations (`op`) are $+, -, *, /, ||, \&\&, <, <=, >, >=, ==, !=$.
- `CONST` is either an integer `(+-)`[0-9]+| or a float number `(+-)`[0-9]*.[0-9]+|.
- `id` is any string consisting of [a-z], [A-Z], [0-9] and _ (the underscore) of any length.
- Boolean binary operations (|| and `&&`) follow the "shortcut" semantics. For example, in `(true` | (a=1; false))|, `a=1` is not executed.
- All variables in a class are taken as "public" variables (as defined in Java).
- A subclass inherits all the methods and variables from its superclass (if any, recursively). To simplify, we assume that a subclass may override methods declared in its ancestor classes, but must have the same signature (i.e., the same list of types, albeit may use different id's). And a variable declared in a subclass cannot use the same id as any variable declared in any of its ancestor classes.

Liveness result of the example

```
Number of iterations=89
----------------------
 (0): 1
def=
use=
In=99, 101,
```

```
Out=99, 101,
----------------------
 (1): 2
def=250,
use=99,
In=101, 99,
Out=250, 101,
----------------------
 (2): 3
def=249,
use=250,
In=101, 250,
Out=249, 101,
----------------------
 (3): 4
def=251,
use=249,
In=101, 249,
Out=101, 251,
----------------------
 (4): 5
def=252,
use=251, 101,
In=251, 101,
Out=252,
----------------------
 (5): 6
def=
use=252,
In=252,
Out=
----------------------
 (6): 7
def=
use=
In=
Out=
----------------------
 (7):
```

```
def=
use=
In=
Out=
```

Interference graph of the example

```
r0,  (0): 102 97 84 79 73 66 68 70 71 54 38 42 20 25 1
s0,  (1):
119,  (2): 1
118,  (3): 1
171,  (4): 1 3
103,  (5): 1
172,  (6): 5 1
120,  (7): 1 5
121,  (8): 5 1
173,  (9): 1 5 8
174,  (10): 1 5
175,  (11): 5 1 10
122,  (12): 1 5
123,  (13): 1 5
124,  (14): 5 1
176,  (15): 1 5 14
177,  (16): 1 5
163,  (17): 1
178,  (18): 1
179,  (19): 1 18
164,  (20): 1
117,  (21): 22 20
180,  (22): 20
125,  (23): 20
181,  (24): 20 23
109,  (25): 20
182,  (26): 20 25
127,  (27): 20 25
183,  (28): 25 20 27
128,  (29): 20 25
126,  (30): 25 20
116,  (31): 20 25
184,  (32): 25 20 31
```

```
108,  (33): 20 25
185,  (34): 33 25 20
129,  (35): 25 33 20
130,  (36): 20 33 25
186,  (37): 20 33
165,  (38): 20
187,  (39): 38 20
132,  (40): 38 20
131,  (41): 42 20 38
105,  (42): 38 20
133,  (43): 42 20 38
110,  (44): 38 20 42
134,  (45): 44 42 20 38
188,  (46): 38 20 42 45 44
135,  (47): 45 44 42 20 38
136,  (48): 38 20 42 44 45
189,  (49): 38 20 42 44
137,  (50): 42 20 38 44
138,  (51): 44 38 20 42
190,  (52): 42 20 38 44 51
191,  (53): 42 20 38 44
106,  (54): 38 20 42
139,  (55): 42 54 20 38
111,  (56): 38 20 54 42
140,  (57): 56 42 54 20 38
192,  (58): 38 20 54 42 57 56
141,  (59): 57 56 42 54 20 38
142,  (60): 38 20 54 42 56 57
193,  (61): 38 20 54 42 56
143,  (62): 42 54 20 38 56
144,  (63): 56 38 20 54 42
194,  (64): 42 54 20 38 56 63
195,  (65): 42 54 20 38 56
107,  (66): 38 20 54 42
196,  (67): 42 66 54 20 38
170,  (68): 92 20 54 66 42
197,  (69): 42 66 54 68 20
169,  (70): 68 54 66 42
147,  (71): 42 54 66 70
```

```
r1,  (72): 97 102 79 84 0 70 68 66 54 42
198,  (73): 70 68 66 42 54
149,  (74): 68 66 42 54 70
199,  (75): 70 54 42 66 68 74
150,  (76): 68 66 42 54 70
200,  (77): 68 66 42 54 70
151,  (78): 54 42 70 66 68
113,  (79): 66 70 42 54 68
201,  (80): 68 54 42 79 70 66
153,  (81): 70 79 42 54 68 66
152,  (82): 83 66 68 54 42 79 70
154,  (83): 70 79 42 54 68 66
155,  (84): 66 68 54 42 79 70
112,  (85): 68 70 54 66 42 79
156,  (86): 42 66 54 70 68 85
202,  (87): 42 66 54 70 68
167,  (88): 70 54 66 42
203,  (89): 42 66 54 88 70
166,  (90): 88 54 66 42
204,  (91): 42 66 54 88 90
168,  (92): 88 54 66 42
205,  (93): 42 66 54 92 88
206,  (94): 42 66 54 68 92
207,  (95): 68 54 42 66 70
157,  (96): 66 42 70 54 68
115,  (97): 54 70 42 66 68
208,  (98): 68 66 42 97 70 54
159,  (99): 70 97 42 66 68 54
158,  (100): 101 54 68 66 42 97 70
160,  (101): 70 97 42 66 68 54
161,  (102): 54 68 66 42 97 70
114,  (103): 68 70 54 66 42 97
162,  (104): 42 66 54 70 68 103
209,  (105): 42 66 54 70 68
210,  (106): 42 66 54 88 70
```