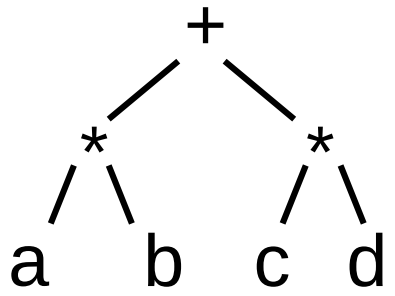
4 Expressions

The next abstraction that we will add to our language are expressions such as `a*b + c*d`. This is a textual representation of the following expression tree that specifies how the expression is to be evaluated:



Later in the course, we will learn how to construct the expression tree from the textual representation, but for now, we will assume that input expressions are already in the form of trees.

To evaluate this expression, we need to multiply the values of a and b, then multiply the values of c and d, and finally add the two products. We will consider three techniques for generating assembly language code to evaluate an expression such as this one. In each case, a key consideration is where to store the temporary results of evaluating each subexpression.

4.1 Technique 1: using a stack

One place where results of subexpressions can be stored is the stack. We can push arbitrarily many values onto the stack, allowing us to evaluate expressions of arbitrary depth. As a convention, we will generate code that evaluates the expression and leaves its value in register 3, `Reg.result`. In general, to evaluate a binary expression $e_1 \odot e_2$, where $\odot$ is any arithmetic operator, we recurse to evaluate e_1, save its value on the stack, recurse to evaluate e_2, pop the value of e_1 off the stack, and apply the arithmetic operator to the two values. The stack is needed so that evaluation of e_2 does not overwrite the result obtained from evaluating e_1. This approach is implemented by the following pseudocode:

```
43 /** Generate assembly language code to evaluate a binary
44  *  expression "e1 operator e2", where e1 and e2 are
45  *  subexpressions and operator is some arithmetic operator,
46  *  and place the resulting value in Reg.result.
47  */
48 def generateCode(e1, operator, e2): Code = block(
49   // code to evaluate subexpression e1
50   // code to push Reg.result onto the stack
51   // code to evaluate other subexpression e2
52   // code to pop top of stack into Reg.scratch
53     // now Reg.scratch has value of e1; Reg.result has value of e2
54   // code to apply operator to Reg.scratch and Reg.result and
55   //   store the result of the operation in Reg.result
56 )
```

This pseudocode generates a block of assembly language code containing the described operations. At lines 49 and 51, the `generateCode` function makes recursive calls to generate assembly language code for the subexpressions.

If we trace the execution on `generateCode` on our example expression `a*b + c*d`, the outer call on the whole expression will make two recursive calls to itself to generate code for the subexpressions `a*b` and `c*d`. The overall output of `generateCode` on the example expression looks as follows (in the form of assembly language pseudocode):

```
57 // generateCode(a*b + c*d)
58
59     // generateCode(a*b)
60     read a into register 3
61     push register 3
62     read b into register 3
63     pop into register 4
64     r3 := r4 * r3
65
66 push register 3
67
68     // generateCode(c*d)
69     read c into register 3
70     push register 3
71     read d into register 3
72     pop into register 4
73     r3 := r4 * r3
74
75 pop into register 4
76 r3 := r4 + r3
```

Exercise 14. *Choose some values for the variables a, b, c, and d, and trace through the execution of the above generated assembly language pseudocode, noting the contents of the stack at each step, to understand how this generated code evaluates the expression.*

This technique for generating code for an expression has the following properties:

- ☺ The technique is *general* in that it can generate correct code for expressions of arbitrary depth. The stack can hold an arbitrary number of results of subexpressions.
- ☹ The generated code contains many pushes and pops on the stack, and is thus not very *efficient* in terms of code size and execution speed.
- ☹ The use of the stack makes the generated code *complicated* and difficult to understand, not only for people, but especially for later passes in the compiler that might optimize or improve the code further.

4.2 Technique 2: using variables (virtual registers)

In the last module, we created variables as an abstraction for areas of memory that can store values. Now that we have this abstraction, we can use it to implement a simpler code

generation technique. If one can use variables and create arbitrarily many fresh variables whenever it is convenient, it is easy to see that the example expression can be evaluated as:

```
v1 := a * b
v2 := c * d
v1 + v2
```

The following pseudocode function `generateCode` will generate code for an expression in this form. The return type of the function is `(Code, Variable)`, which indicates that the function returns a pair of a piece of assembly language code and a variable. The function first recurses on each of the subexpressions `e1` and `e2`; for each, we generate code to evaluate it and a variable that holds the resulting value. Then (in line 86), we just apply the arithmetic operator to the two variables, storing the result in a freshly created variable `v3`. Finally (in line 88), we return the code to evaluate the subexpressions and the arithmetic operator, together with the variable `v3` that holds the result.

```
77 /** Generate assembly language code to evaluate a binary expression
78  *  e1 operator e2, where e1 and e2 are subexpressions and operator
79  *  is some arithmetic operator, and place the resulting value
80  *  in a variable.
81  */
82  def generateCode(e1, operator, e2): (Code, Variable) = {
83   (c1, v1) = // code/variable to evaluate subexpression e1
84   (c2, v2) = // code/variable to evaluate other subexpression e2
85   v3 = new Variable
86   code = c1 ++ c2 :+ // code to apply operator to v1 and v2
87                      //   and store result in v3
88   (code, v3)
89 }
```

Tracing the above `generateCode` function on our example expression `a*b + c*d` generates the following assembly language pseudocode:

```
90 // generateCode(a*b + c*d)
91
92     // generateCode(a*b)
93     v1 := a * b
94
95     // generateCode(c*d)
96     v2 := c * d
97
98 v3 := v1 + v2
```

Notice how simple this code is, and how close it is to the obvious steps to evaluate the expression. Of course, to turn this code into assembly language, we need to translate the uses of variables into direct accesses of physical memory or registers. We saw one easy way to implement variables in the last module. A compiler that generates many variables and many uses of variables should probably use a more sophisticated and more efficient method, as we will see at the end of this module.

This technique for generating code for an expression has the following properties:

☺ The generated code is simple and flexible, and easy for a compiler to improve and optimize further.

☹ Machine language arithmetic instructions operate on registers, not directly on variables, so we cannot directly express expressions such as `a*b` or `v1 + v2` as a single machine language instruction.

☹ To further translate this code into efficient assembly language, we need a way to map variables to registers, rather than to stack locations, and to map a large number of variables to a limited number of registers.

A technique that alleviates the last two disadvantages is called *register allocation*, and we will discuss it at the end of this module. We could implement register allocation as an assignment in this course, but it would interact in complicated ways with future assignments, so we will not do so. Instead, in Assignment 4, you will implement a third code generation technique that is a compromise between Techniques 1 and 2. This third technique alleviates the first disadvantage (that machine language arithmetic instructions do not work directly on variables), although it does not address the second disadvantage (efficiency of the generated assembly language).

4.3 Technique 3: variables for temporary values, arithmetic operations on registers

In the third technique, we will return to the convention that the code generated to evaluate an expression leaves the resulting value in register 3, `Reg.result`, rather than in a variable. This is because the machine language arithmetic instructions require the values to be in registers, not variables. However, instead of using the stack to store the temporary values, we will use the variable abstraction, creating a fresh variable to store the value of each subexpression. The pseudocode for the third code generation technique is shown below. Notice the similarity with the first technique.

```
99  /** Generate assembly language code to evaluate a binary
100  *  expression "e1 operator e2", where e1 and e2 are
101  *  subexpressions and operator is some arithmetic operator,
102  *  and place the resulting value in Reg.result.
103  */
104 def generateCode(e1, operator, e2): Code = {
105   t1 = new Variable
106   block(
107     // code to evaluate subexpression e1
108     // code to write Reg.result into variable t1
109     // code to evaluate other subexpression e2
110     // code to read variable t1 into Reg.scratch
111       // now Reg.scratch has value of e1; Reg.result has value of e2
112     // code to apply operator to Reg.scratch and Reg.result and
```

```
113     //   store the result of the operation in Reg.result
114   )
115 }
```

Tracing the execution of this technique on the example expression `a*b + c*d` results in the following generated code.

```
116 // generateCode(a*b + c*d)
117
118     // generateCode(a*b)
119     read a into register 3
120     write register 3 into variable t2
121     read b into register 3
122     read variable t2 into register 4
123     r3 := r4 * r3
124
125 write register 3 into variable t1
126
127     // generateCode(c*d)
128     read c into register 3
129     write register 3 into variable t3
130     read d into register 3
131     read variable t3 into register 4
132     r3 := r4 * r3
133
134 read variable t1 into register 4
135 r3 := r4 + r3
```

Notice the similarity between this code and the code generated by Technique 1; in particular, it is equally inefficient. In a real-world compiler that needs to generate efficient code, we would prefer Technique 2 combined with a good register allocator to generate more efficient code.

In the course, we do not wish to implement a register allocator to avoid complicating future assignments. Technique 3 is a compromise with the following advantages:

- Compared to Technique 1, the code generated by Technique 3 is easier to understand and would be easier to optimize further because the value of each subexpression is stored in its own variable, rather than using different stack locations hold the values of various subexpressions at various times.
- Compared to Technique 2, the code generated by Technique 3 can be translated directly to assembly language without *requiring* a register allocator.

Even though a real-world compiler would implement Technique 2 for efficiency, in this course, in Assignment 4, you will implement Technique 3, to avoid the need to implement register allocation.

4.4 Keeping track of temporary variables

When we implemented a frame in Assignment 3 to hold the values of variables, we needed a list of all the variables that are used in the code. In the code that we generate to implement expressions, we create new temporary variables at various times. Eventually, we will need a list of all of these variables so we can create a frame with enough space for them.

One way to keep track of temporary variables is with a new kind of `Code` called a `Scope`. A `Scope` defines one or more new variables and groups them together with a fragment of code in which those variables can be accessed. By creating a `Scope` for every temporary variable that we create, we can easily make a list of all the temporary variables by finding all the `Scope`s in the generated code (using the `transformCode` method from Assignment 2) and reading the variable lists from them. You will implement this in the `eliminateScopes` method in Assignment 4.

The pseudocode for Technique 3 for evaluating expressions can be adapted to generate `Scope`s to keep track of the temporary variables that it creates as follows:

```
136 /** Generate assembly language code to evaluate a binary
137  *  expression "e1 operator e2", where e1 and e2 are
138  *  subexpressions and operator is some arithmetic operator,
139  *  and place the resulting value in Reg.result.
140  */
141 def generateCode(e1, operator, e2): Code = {
142   t1 = new Variable
143   Scope(t1,
144     block(
145       // code to evaluate subexpression e1
146       // code to write Reg.result into variable t1
147       // code to evaluate other subexpression e2
148       // code to read variable t1 into Reg.scratch
149       // now Reg.scratch has value of e1; Reg.result has value of e2
150       // code to apply operator to Reg.scratch and Reg.result and
151       //   store the result of the operation in Reg.result
152     )
153   )
154 }
```

4.5 Control structures

In addition to expressions, in Assignment 4, you will also compile control structures: if statements and while loops. Both of these require compiling a conditional expression and arranging conditional and unconditional branch instructions around the code inside the control structure. In pseudocode, the translation of an if statement `if( e1 op e2 ) T else E`, where `op` is a comparison operator, is as follows:

```
155 /** Generate assembly language code to implement the if statement
156  *  if( e1 op e2 ) T else E.
```

```
157   */
158 def generateCode(e1, operator, e2, T, E): Code = {
159   t1 = new Variable
160   elseLabel = new Label
161   endLabel = new Label
162   Scope(t1,
163     block(
164       // code to evaluate subexpression e1
165       // code to write Reg.result into variable t1
166       // code to evaluate other subexpression e2
167       // code to read variable t1 into Reg.scratch
168       // now Reg.scratch has value of e1; Reg.result has value of e2
169       // code to apply operator to Reg.scratch and Reg.result and
170       //   branch to elseLabel if the result of the comparison is false
171       //   (using beq or bne)
172       T
173       beq(Reg.zero, Reg.zero, endLabel)
174       Define(elseLabel)
175       E
176       Define(endLabel)
177     )
178   )
179 }
```

Notice that the first half of this code, which evaluates the conditional expression `e1 op e2`, is very similar to the code for evaluating other binary expressions that we discussed earlier in this module.

If the conditional expression evaluates to false, the generated code skips over the then-block `T` to the `elseLabel`. If the condition evaluates to true, execution of the generated code falls through the conditional branch into the then-block `T`. At the end of the then-block `T`, an unconditional branch (that tests whether $0 = 0$) skips over the else-block `E` to the `endLabel`.

The code generated for a while loop is similar and is left as an exercise for you to work out yourself, using your existing understanding of how while loops should work. In fact, the code to implement a while loop is slightly simpler than that for an if statement because a while loop has only a single body, rather than the two branches of an if statement.

4.6 Arrays

Many high-level programming languages provide *arrays* as an elementary data structure. An *array* is a collection of elements indexed by consecutive natural numbers. The value stored in each element can be read and written efficiently using the index of the element.

Arrays can be implemented in machine language by using a contiguous range of memory addresses to store the values of the array elements. Each element of the array is assigned the same amount of memory; for example, if the array elements are 32-bit integers, each element could be one word, or four bytes. The elements are stored contiguously in order of their index, so it is straightforward to calculate the memory address of any element of the

array given its index i. We multiply i by the size (in bytes) of each array element to get the distance from the beginning of the array to the ith element. We then add this distance to the memory address of the beginning of the array to get the memory address at which the value of the ith element is stored. Finally, the value of the array element can be read or modified using a `LW` or `SW` instruction with that address.

The arithmetic expressions covered earlier in this module are sufficient to express the calculation of the address of a specified array element. To implement a read or write of the element at that address, we need only to add operations that generate the necessary `LW` and `SW` instruction. In Assignment 4, the `deref` method generates `Code` that evaluates an expression to yield a memory address, then reads the value stored at that address. The `assignToAddr` method generates `Code` that evaluates two expressions, one to yield the memory address to which a value is to be written and the other to yield the value to be written to that address, and then writes the value to that memory location.

4.7 Register allocation

In the previous module, we defined a simple implementation of local variables: each variable is assigned a unique offset in the current stack frame. Although this scheme is simple, the generated code is not as efficient as it could be: each time we wish to use a variable in an expression, we need a `LW` instruction to read the value of the variable from memory into a register. After we evaluate an expression, we need a `SW` instruction to store the resulting value back into a variable. This inefficiency is especially pronounced in code that uses variables heavily, such as the expression evaluation strategy in Technique 2 above, which uses a variable for the result of every subexpression of an expression. We could generate more efficient code, avoiding the memory loads and stores, if variables were stored directly in registers, rather than on the stack.

The purpose of *register allocation* is to determine which variable should be stored in which register. Register allocation is necessary because the CPU provides a fixed number of registers, but a procedure may use an arbitrary number of variables. For efficiency, we would like as many variables as possibly to be stored in registers, rather than in stack offsets. One simple strategy would be to greedily allocate variables to registers on a first-come first-served basis until we run out of registers. For example, if we have 25 registers available to store variables, the first 25 variables we see would be stored in registers and all remaining variables would be in stack offsets. It turns out, however, that in many cases, multiple variables can *share* the same register, so it is possible to allocate many more than 25 variables to 25 registers.

To see this, consider the following example expression: `a + b + c + d + e`. Writing assembly language code by hand, you might choose some register as an accumulator, initialize it to zero, then add `a` to it, then add `b` to it, and so on up to `e`, until that one register contains the final sum. Technique 2 above would generate the following code in terms of temporary variables:

```
180 t1 := a + b
181 t2 := t1 + c
182 t3 := t2 + d
```

```
183 t4 := t3 + e
```

Although this code uses four variables, it will work correctly if all four variables share the same register, corresponding to the chosen accumulator register in the solution implemented by hand:

```
184 r1 := a + b
185 r1 := r1 + c
186 r1 := r1 + d
187 r1 := r1 + e
```

On the other hand, consider the code generated by Technique 2 for our earlier example expression `a*b + c*d`:

```
188 t1 := a*b
189 t2 := c*d
190 t3 := t1 + t2
```

In this case, if we share one register for both `t1` and `t2`, then the result of evaluating `c*d` will overwrite the result of evaluating `a*b`, so it is *not* safe to share the register in this case.

To determine when variables can be shared, we define the notion of liveness: a variable v is *live* at program point p if the value stored in v at p may be read from v sometime after executing p. In a sequence of instructions, a program point is the location in between any two consecutive instructions. For example, at the point just after line 188, variable `t1` is live because its current value will be read in line 190. Similarly, at the point just after line 189, both `t1` and `t2` are live because their current values will be read immediately in the next line, line 190. Given a variable v, the *live range* of v is the set of all program points p at which v is live. Then two variables can share the same registers if their live ranges are disjoint, that is, if they are never both live at the same program point. As an analogy, two co-op students can share the same rented room if they are off-stream from each other, so they are never both on campus at the same time.

It can be convenient to think about which variables are not live, or *dead*. A variable can be dead for one of two reasons:

1. A variable is dead if it will never be read, if there are no reads of the variable after the current program point.

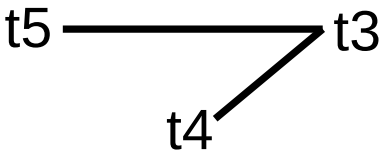
2. A variable is dead if its current value will be overwritten before it is read, if every execution path from the current program point to a read of the variable passes through a write of the variable.

Exercise 15. *At each program point in the following sequence of instructions, write the set of variables that are live at that point. Include only the temporary variables whose names start with* `t`*, ignoring the variables* `a` *to* `g`*.*

```
191 t1 := a*b
192             live set: { _______ }
193 t2 := c*d
```

```
194             live set: { _______ }
195 t3 := t1 + t2
196             live set: { _______ }
197 t4 := e*f
198             live set: { _______ }
199 t5 := t3 - t4
200             live set: { _______ }
201 g := t3 + t5
```

After we have computed live sets at every program point, we can summarize the information in an *interference graph*, which has a vertex for each variable, and an edge connecting a pair of variables if they *interfere* in that they are live at the same time. Assuming you determined the correct live sets in the exercise, they should yield the following interference graph:



Then, a valid register allocation corresponds to a *colouring* of the interference graph. A *graph colouring* is an assignment of colours to the vertices of a graph such that every edge connects two vertices with different colours. Colours correspond to registers: every edge (indicating that two variables are live at the same time) connects vertices (variables) with different colours (assigned to different registers). We usually want a minimal colouring (with the minimum possible number of colours), corresponding to a register assignment using the minimum number of registers.

For example, a minimal colouring assigns the following colours (registers) to our example interference graph: `t1` and `t3` are assigned register 1, and `t2`, `t4`, and `t5` are assigned register 2. We then replace each occurrence of a variable in the code with its assigned register, yielding the following instructions that use the two registers rather than the five variables:

```
202 r1 := a*b
203 r2 := c*d
204 r1 := r1 + r2
205 r2 := e*f
206 r2 := r1 - r2
207 g := r1 + r2
```

Exercise 16. *1. Check that this colouring of the interference graph is a valid colouring, in that each edge connects two vertices with different colours.*

2. *Convince yourself that this colouring is minimal, that the graph cannot be coloured with fewer than two colours.*

3. *Convince yourself that the sequence of instructions in terms of two registers above computes the same result as the sequence of instructions in terms of five temporary variables given in the previous exercise.*

Deciding whether an *arbitrary* graph can be coloured with k colours is an NP-complete problem. NP-completeness is covered in CS 341. If a problem is NP-complete, then a polynomial time algorithm solving that problem could be modified to solve all other NP-complete problems in polynomial time. There is no known polynomial time algorithm for finding a minimal graph colouring, and if one were to be found, it would be a major breakthrough in solving many other important problems. Therefore, there is no known polynomial time algorithm for optimal register allocation.

Nevertheless, there are heuristic algorithms that find small graph colourings in many practical cases, and they are frequently used for register allocation in practical compilers. Furthermore, many interference graphs that occur in compilers are not arbitrary, but have special structure that makes it possible, even quite easy, to find minimal colourings and thus optimal register allocations. If you are interested in further reading, you can read more about this in the following paper: Sebastian Hack, Daniel Grund, and Gerhard Goos: Register Allocation for Programs in SSA-Form. Compiler Construction, 15th International Conference, CC 2006.