Lecture 14: Code generation for expressions

David Hovemeyer

October 18, 2023

601.428/628 Compilers and Interpreters

Agenda

- Code generation for expressions
- Variables and storage
- Implicit (and explicit) conversions
- Assignments
- Pointers and pointer operations
- Function calls

Code generation for expressions

Code generation for expressions

- Expression: construct which computes a value
- In C: could also assign to a variable, but let's wait to discuss that possibility
- We'll focus on generating high-level IR code (RISC-like, infinite number of "virtual" registers)
- This is surprisingly easy!

Something important to understand

The goal of initial code generation (from the annotated AST) is to generate high-level code that is *correct*, *consistent*, and *straightforward*.

Efficiency is not the main concern at this point.

Inefficiencies can be addressed later on when performing code optimization.

Code generation via recursive treewalk

- Generating code for an expression is two distinct but related tasks:
  1. Generating a sequence of zero or more instructions which, when executed, will compute the result value of evaluating the expression
  2. Determining where the result of the evaluation is stored
- Task 1 is accomplished by recursively visiting subexpressions (if necessary), then emitting instruction(s) as needed to "complete" the computation
  - Literal values and variable references are base cases in this recursion
- Task 2 is accomplished by allocating a temporary virtual register to hold the result of the evaluation: the emitted code will place the result value there
  - Suggestion: each AST node contains an `Operand` specifying the location where the evaluation result is stored (normally a virtual register)

Code generation pseudo-code

```
CodeGen(ast) {
    if (ast→tag = LITERAL) {
        ival = make_immediate(ast→get_ival())
        dest = alloc_tmp_vreg()
        emit(HINS_mov, dest, ival)
        ast→set_operand(dest)
    } else if (ast→tag = VARREF) {
        val = get_var_storage_loc(ast→get_sym())
        ast→set_operand(val)
    } else if (ast→tag = BINOP) {
        opcode = get_binop_opcode(ast→get_kid(0))
        left = ast→get_kid(1), right = ast→get_kid(2)
        CodeGen(left)
        CodeGen(right)
        dest = alloc_tmp_vreg()
        emit(opcode, dest, left→get_operand(), right→get_operand())
        ast→set_operand(dest)
    } else ...other cases...
}
```

Some observations

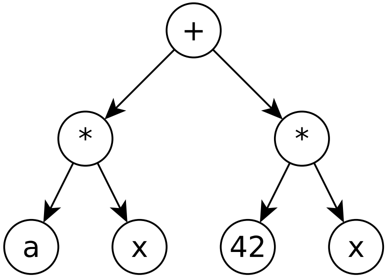
- The code could also be implemented as an AST visitor (which is how we recommend that you implement high-level codegen in Assignment 4)
- The make_immediate() function yields an immediate operand (i.e., a literal value)
- The get_var_storage_loc() function yields an operand referring to the storage location for a variable (given a pointer to its symbol table entry)
  - More about this soon
- The alloc_tmp_vreg() function allocates a “temporary” virtual register to put the computed value in (remember that there are an “infinite” number of virtual registers)
- The AST node is annotated with an operand indicating where the result of the evaluation is (i.e., the temporary vreg)

An example

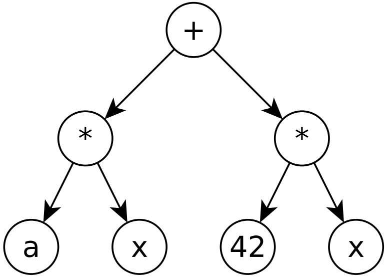
```
int poly(int x, int a) {
  return a * x + 42 * x;
}
```

Assume:

- x is vr10
- a is vr11
- Allocation of temporary vregs starts with vr12

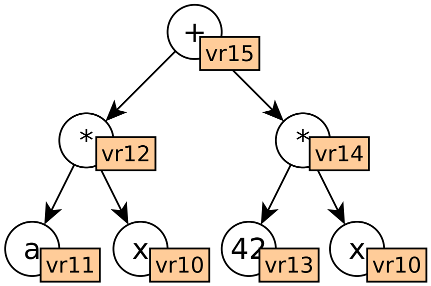


An example



x is vr10, a is vr11, temp vreg allocation starts with vr12

An example



```
mul_l    vr12, vr11, vr10
mov_l    vr13, $42
mul_l    vr14, vr13, vr10
add_l    vr15, vr12, vr14
```

`x` is `vr10`, a is `vr11`, temp vreg allocation starts with `vr12`

Discussion

- Code generation for variable references doesn't require any instructions if the variable is a vreg
- The code generator needs to be aware of operand sizes; e.g., for 32-bit `int` operand values, the "`_l`" suffix (e.g., `mov_l`) is used
  - The semantic analyzer will have annotated each expression node with a type, so the code generator can just ask the type for its storage size

Variables and storage

What is a variable?

A variable is a storage location in which a value belonging to the variable's data type may be stored.

Variables can be referred to in the program in various ways:

- Normal "named" variable: `x`, `p`, `foo`, etc.
- Array element: `a[0]`, `bar[42]`
- Field of struct instance: `p.x`

Storage

A variable's storage can be provided in two ways.

If a variable's type is has sufficiently small representation (generally, 8 bytes or less), its storage can be a register.

If a variable's type requires a representation whose size exceeds what will fit in a register, its storage must be in main memory.

Also, C generally[1] requires memory storage for

- Arrays
- Static or global variables (of any size)
- Instances of struct types
- A variable whose address is taken (meaning a pointer to that variable's storage exists)

[1] A really clever compiler might be able to use register storage for some of these in some cases.

Implications for code generation

Since the high-level code generator will need to have a way of generating code for variable references, it will need to have some notion of storage allocation for variables.

This initial storage allocation will need to be done in such a way that later on, when low-level (target) code is generated, accesses to registers and memory work correctly.

- For example, variables allocated in memory should respect alignment requirements
- For this reason, high-level code might not be *completely* independent of the target architecture

Suggested approach

You might approach the problem of allocating storage for variables as follows:

- For each local variable whose storage could be a virtual register, allocate a specific virtual register for that variable
- For arrays, struct instances, and any variable whose address is taken, allocate a chunk of memory with the appropriate size and alignment in the stack frame
    - The high-level `enter` and `leave` instructions are used to allocate and deallocate memory in the stack frame being used for local variables
    - The high-level `localaddr` instruction computes the address of memory at a specific offset in the current stack frame and places it in a virtual register

Note that we neither require nor expect your compiler to handle global variables (although these aren't difficult.)

But there aren't an infinite number of registers!

- The high-level code generator assumes an unlimited supply of virtual registers
- But actual CPUs have only a limited number of registers
- This isn't really a problem!

Why we don't need to worry (reason #1)

Just because the high-level code generator calls a location a "virtual register" doesn't mean that location needs to be a CPU register.

It could be a memory location in the stack frame.

For the initial code generator(s) you implement in Assignment 4, we recommend just allocating storage for all virtual registers in the function's stack frame (alongside any variables requiring memory storage.)

This approach will be slow, but it is easy and it will work.

Why we don't need to worry (reason #2)

Looking ahead:

- Actual CPU registers will only be needed for "live" values during computations: most virtual registers store a live value during a very brief timespan (usually just a few instructions)
  - Once the value in a register "dies" (is no longer needed), its CPU register can be used for a different value
  - So, a small number of CPU registers is usually sufficient for live values
- In Assignment 5, you can use *register allocation* to use CPU registers as needed for live values
  - You can also decide to allocate CPU registers for some local variables

Recording decisions about storage allocation

We recommend that you use each variable's symbol table entry to store information about what the variable's storage location is.

This will generally be either

1. A virtual register (e.g., `vr10`)
2. An offset within the memory in the stack frame (e.g., 16)

To create an `Operand` for this information: if the storage is a vreg, this is trivial (e.g., operand is `vr10`.) If the storage is in memory, code may be required to load the variable's memory address into a vreg:

```
localaddr vr15, $16 /* operand (vr15) refers to the variable */
```

Implicit (and explicit) conversions

Conversions

At some points in the program execution, it will be necessary to convert a value from one type to another.

This may be necessary because of implicit conversion rules, or explicit conversions (type casts) in the code.

Explicit conversions appear directly in the AST (i.e., `AST_CAST_EXPRESSION` nodes.)

It's not a bad idea to represent implicit conversions explicitly in the AST as well. (The semantic analyzer can add them.)

Implicit conversion example

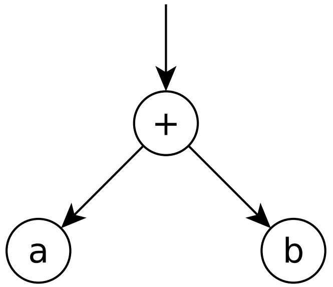
Simple example:

```
short example(char a, char b) {
  return a + b;
}
```

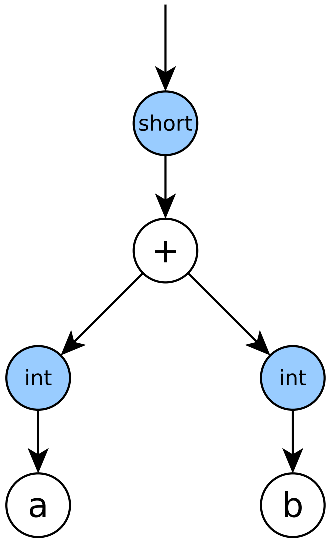
C requires all integer operations to be performed with `int` as the minimum precision. Also, the resulting `int` sum must be converted to `short` (the return type.) So this program is equivalent to

```
short example(char a, char b) {
  return (short) ((int) a + (int) b);
}
```

Representing implicit conversions in the AST



$\Rightarrow$



Code generation for integer conversions

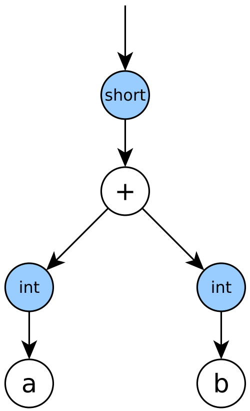
A conversion of an integer value could be a *promotion* or a *truncation*:

- Promotion: less precise type converted to more precise type
- Truncation: more precise type converted to less precise type

For integer promotions, use the `sconv` or `uconv` high-level instructions depending on whether the value being converted is signed or unsigned. E.g., `sconv_bl` to convert a `char` (8 bit signed integer) to `int` (32-bit signed integer.)

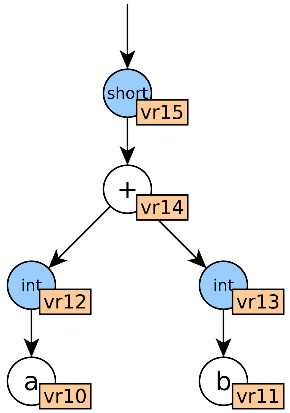
Nothing special needs to be done for a truncation: just use a `mov` with an operand size appropriate for the converted-to type. For example, use `mov_w` to truncate an `int` value to a (16-bit) `short` value.

An example



Assume a is `vr10`, b is `vr11`

An example



Assume a is `vr10`, b is `vr11`

```
sconv_bl vr12, vr10
sconv_bl vr13, vr11
add_l    vr14, vr12, vr13
mov_w    vr15, vr14
```

Assignments

Assignments

An *assignment* to a variable stores a computed value in that variable. This is as simple as emitting a `mov` instruction (of the appropriate size) to copy the computed value from its location (usually a temporary vreg) to the variable's storage location.

C has the additional quirk that assignments *yield* a value. For example:

```
int a, b;
a = (b = 3); // assigns 3 to b, then assigns 3 to a
```

lvalues

An expression is an *lvalue* if it has a (potentially) assignable storage location. Variables (including array elements and struct fields) are lvalues. Dereferenced pointers are also lvalues.

So, your compiler should be prepared to handle any assignment where the subexpression on the left-hand side is an lvalue.

Memory references

The "assembly" notation for high-level code uses a similar convention to indicate a memory reference as "AT&T style" x86-64 assembly language:

- `vr10` refers to virtual register 10
- `(vr10)` refers to the memory location that the pointer value in `vr10` points to

Easy case: variable's storage is a vreg

If a variable's storage is a virtual register, an assignment is trivial: use a `mov` to copy the computed value (the right-hand side of the assignment) to that vreg.

```
/* C code */
int x;
x = 2 + 3;
```

```
/* translation (x is vr10) */
mov_l     vr11, $2
mov_l     vr12, $3
add_l     vr13, vr11, vr12
mov_l     vr10, vr13
```

Harder case: variable's storage is in memory

If a variable's storage is in memory, an assignment must

1. Load the variable's address into a vreg
2. Move the computed value to the location the vreg points to

```
/* C code */
int x;
int *p;
p = &x;     // force x to have
            // memory storage
x = 2 + 3;
```

```
/* translation (p is vr10,
   x is at offset 0 in
   local storage) */
localaddr vr11, $0
mov_q     vr10, vr11
mov_l     vr12, $2
mov_l     vr13, $3
add_l     vr14, vr12, vr13
localaddr vr15, $0
mov_l     (vr15), vr14
```

Pointers and pointer operations

Pointers, address-of, dereference

- A pointer is the address of a storage location in memory
- The address-of operator (`&`) can be applied to any lvalue to yield a pointer to the lvalue's memory location
- The dereference operator (`*`) can be applied to any pointer to refer to the storage of the lvalue that the pointer points to
- This sounds complicated, but there's a really simple way to think about these operations!

An example

Let's say that `x` is an `int` variable whose storage is allocated in memory, at offset 16 in the stack frame's local variable storage.

If we emit the instruction $\boxed{\texttt{localaddr vr15, \$16}}$, then `vr15` is a pointer to `x`'s storage in memory.

That means that the memory reference operand `(vr15)` is synonymous with `x`.

More generally, any lvalue can be described by an operand (vr$N$), where vr$N$ is the virtual register containing a pointer to the lvalue's memory location.

Address-of

The address-of operator (`&`) converts an lvalue to a pointer.

If the variable `x` is represented by the operand `(vr15)`, then `vr15` is a pointer to `x`. More generally, address-of is the conversion

$$(\texttt{vr}N) \Rightarrow \texttt{vr}N$$

Dereference

The dereference operator (*) converts a pointer to an lvalue.

If `vr15` is a pointer to the variable `x`, then the operand `(vr15)` refers to the variable `x`. More generally, dereference is the conversion

$$\mathtt{vr}N \Rightarrow (\mathtt{vr}N)$$

Pointer arithmetic

Pointer arithmetic involves adding an offset to a base pointer to compute the address of an array element some number of elements displaced from where the base pointer is pointing.

Array subscript operations are equivalent to dereferencing the result of a pointer arithmetic computation.

More about these when we cover code generation for arrays.

Function calls

Function calls

In the high-level IR,

- `vr1` through `vr9` are argument registers
- `vr0` is the return value register

So, when calling a function:

1. Generate code to evaluate argument expressions, and store the computed values in the appropriate argument register
2. Emit a `call` instruction to call the named function
3. When the call returns, the return value is in `vr0`

Non-leaf functions

A function which does not call other functions is a *leaf function*.

When generating code for a non-leaf function (which *does* call other functions), the argument registers must be used for both receiving argument values from the caller and passing argument values to callees.

To make this work, at the beginning of a non-leaf function, each argument register value should be copied to the storage for the corresponding parameter variable (whose storage should be allocated the same way as any other local variable.) Note that there is no harm in also doing this for leaf functions.

This approach makes all of the argument registers available for passing values to called functions.