

ASSIGNMENT #5: Working on your Quads

Not surprisingly, the goal of Assignment #5 is to emit quads.

Now, there is no particular standard way of encoding or representing quads. We've used certain notation in class and in the lecture notes, but you are not bound to that. You are not bound to the algorithms and methods presented either.

The only requirement is that you generate quads and output them in a human-readable form, and that you use notation which makes sense, keeping in mind that you will soon use these quads to generate actual assembly code.

THINGS YOU DO AND DO NOT HAVE TO DO:

- + Expressions
 - + Operators
 - + arithmetic operators including pointer add/subtract
 - + sizeof (you need that anyway for pointer arith)
 - + assignment
 - bitwise/shift operators optional but really not that hard
 - ternary and comma operators optional
 - no casts
 - no C99 features such as compound literals
 - + Support operations on ints and pointers
 - You don't have to worry about floating point, different size integer operations, etc.
 - + You do need to get pointer +/- integer and pointer - pointer right. Without this arrays don't work.
 - structure member operators . and -> are optional
 - Implicit in handling structures is assigning offsets to the members. You should assume the size and alignment constraints of your target architecture (perhaps encode them in a table) and then use the algorithm presented in class to lay out the structure.
 - H&S and the C standard also discuss this.
 - + You need to support making a function call, including passing arguments. It's OK to assume that the left operand to a function call expression is a simple identifier (rather than an arbitrary expression of type pointer to fn) if that makes it easier for you.
- + Statements
 - + compound statements
 - Just generate the quads for each statement in series
 - + Expression statements
 - An expression statement that has no "useful" effect or side-effect, e.g. `a+1;` should generate a warning but this is optional

- + If statements
- + Loop statements
 - only one type of loop required, but once you've done one, the others are pretty easy anyway.
- + break/continue statements
 - You need to keep the break and continue basic block targets as part of your "cursor" or "state" while generating quads
- + return statements
 - OK to support just functions returning int, or void
- Goto and labeled statements optional
 - As discussed in class, a labeled statement creates a new basic block and a symbol table entry. The basic block number can be the primary attribute of the entry. Goto statements to a forward label install a symbol table entry with the basic block number undefined. When the matching label is found, find it in the symbol table and assign it a basic block number. If, when quads are being generated, a GOTO statement refers to a label with no defined basic block, that's an error.
- Switch statements optional
 - These are a big pain. Lecture notes discuss three ways of handling them. The repeated compare-and-branch approach is the easiest, should you decide to do this optional part.
- Initialized declarators
 - Still optional. Simple declarations, e.g.
 int a=1;
are straightforward. Complex initializers with all the curly braces could be a big time sink.
- Function formal parameters
 - You need to support function definitions. They don't need to take any parameters, although that would be a nice bonus.
- Optimizations
 - It is not necessary to try to output the "best" series of quads. You might find that your algorithm generates extraneous steps, e.g. taking a value from one place, moving it to another, and moving it back to the first place. Or you might notice that you are performing computations in the target code which could be done at compile-time and replaced with a constant, e.g.
 a=3+4;

In "theory", the architecture-independent optimizer would find these and optimize them. Of course we won't be doing that...

Input:

```

int a;

int f(b)
{
    b=a+3;
}

int g()
{
    struct x {
        int a1;
        int b2;
    } x;
    x.b2=a;
}

```

Output:

AST Dump for function f:

```

LIST {
  ASSIGNMENT
    stab_var name=b def @<stdin>:4
  BINARY OP +
    stab_var name=a def @<stdin>:1
    CONSTANT: (type=int)3
}

```

f:

```

.BB1.1
    b{param} = ADD    a{global},3
    RETURN

```

AST Dump for function g:

```

LIST {
  ASSIGNMENT
    SELECT, member b2
    stab_var name=x def @<stdin>:13
    stab_var name=a def @<stdin>:1
}

```

g:

```

.BB2.1
    %T00001 = LEA    x{lvar}
    %T00002 = ADD %T00001,4
    STORE a{global},[%T00002]

```

RETURN

Discussion: My quad gen output includes the assignment #4 AST dump output too. This is pretty handy for debugging.

By definition, a Basic Block is a potential branch target. It is started by a label, contains no additional labels within, and is terminated by a branch or exit. I am giving my basic blocks labels of the form

.BB.F.N

where F is a counter which increments with each function, and N is the basic block number within the function. I'm thinking ahead to Assignment #6, where those labels will become assembly language labels, and will need to be unique within the entire output, not just unique within a single function.

You have some flexibility in how to present the quad operands in your output. Internally, you have to hold on to the symbol table entries unless you want to start exposing the mechanics of local variable stack frame access now. What I've done above is give a clue as to the nature of the variable (e.g. {lvar} for local; or {global}).

You may also note that I am giving temporaries names which look like registers. Quads are a theoretical, idealized assembly language where there is an infinite supply of "virtual registers" to hold temporary/ intermediate values. During final code generation, this infinite supply will need to be reduced to the finite set of registers found on the target.

I have chosen to expose the access to structure members with the required memory address computations and load/stores. This is in concordance with the philosophy of avoiding complex addressing modes for quad operands, and of eliminating C language typing issues. Other IR schema, such as LLVM, do things very differently, by carrying through the idea of the struct type to the IR level.

I have not exposed the mechanism for accessing local variables, because this is target-specific. E.g. on X86-32 it will involve an offset from the %ebp register, on SPARC it will be the %fp register. When these quads are transformed into assembly (Assignment #6) the LEA of the local var will be translated appropriately. Likewise, accesses to function parameters remain abstracted at this stage, and vary wildly by target.

=====

EXAMPLE 2

=====

Input:

```
int a[10];

f()
{
    int x,*p;
        x=a[3];
        p= &a[5];
}
```

Output:

```
AST Dump for function f:
LIST {
```

```

    ASSIGNMENT
    stab_var name=x def @<stdin>:5
    Deref
    BINARY OP +
    stab_var name=a def @<stdin>:1
    CONSTANT: (type=int)3
    ASSIGNMENT
    stab_var name=p def @<stdin>:5
    ADDRESSOF
    Deref
    BINARY OP +
    stab_var name=a def @<stdin>:1
    CONSTANT: (type=int)5
  }
f:
.BB1.1
    x{lvar} = LOAD      [$a+12]
    p{lvar} = MOV $a+20
    RETURN

```

Discussion:

I have replaced the [] array subscripting operator in the AST with the equivalent pointer expression, as defined in the C standard. This simplifies the number of cases for which the quad generator needs to be coded.

The expression \$a+12 is an example of an address constant, i.e. a global symbol plus or minus an integer offset. There is a further constant folding optimization in that the computation of the offset of the array element (3*4) has been replaced with the constant 12. You are not required to perform either of these optimizations. Without them, the multiply and add would appear explicitly:

```

f:
.BB1.1
    %T00001 = MUL    3,4
    %T00002 = LEA a{global}
    %T00003 = %T00001,%T00002
    x{lvar} = LOAD      [%T00003]
    %T00004 = MUL    5,4
    %T00005 = LEA a{global}
    p{lvar} = ADD %T00004,%T00005
    RETURN

```

EXAMPLE 3

Input :

```

z ()
{

```

```

int a,b;
    if (a<b)
        a=1;
    if (b!=a)
        b=3;
    else
        a=3;
}

```

Output:

```

AST Dump for function z:
LIST {
  IF:
    COMPARISON OP <
      stab_var name=a def @<stdin>:3
      stab_var name=b def @<stdin>:3
    THEN:
      ASSIGNMENT
        stab_var name=a def @<stdin>:3
        CONSTANT: (type=int)1
    IF:
      COMPARISON OP !=
        stab_var name=b def @<stdin>:3
        stab_var name=a def @<stdin>:3
    THEN:
      ASSIGNMENT
        stab_var name=b def @<stdin>:3
        CONSTANT: (type=int)3
    ELSE:
      ASSIGNMENT
        stab_var name=a def @<stdin>:3
        CONSTANT: (type=int)3
  }
z:
.BB1.1
    CMP    a{lvar},b{lvar}
    BRGE   .BB1.3,.BB1.2

.BB1.2
    a{lvar} = MOV 1
    BR     .BB1.3

.BB1.3
    CMP    b{lvar},a{lvar}

```

```

        BREQ      .BB1.5, .BB1.4
.BB1.4
        b{lvar} = MOV 3
        BR        .BB1.6

.BB1.5
        a{lvar} = MOV 3
        BR        .BB1.6

.BB1.6
        RETURN

```

Discussion: I have chosen to represent control flow among basic blocks by explicitly listing the true and false branch targets for each branch. It is not required that you do it this way. You could also generate quads in which only one branch target is given, and it is implied that if the branch is false (not taken) that control falls-through to the next basic block. When using the 2-target approach, the order in which the basic blocks will eventually be output into the assembler is not pre-determined, but will be decided by walking the control flow graph. In the example above, since BB1.2 is the false target of BB1.1, BB1.2 would be output immediately after BB1.1. The unconditional branches to the next BB would also be dropped out, e.g. the BR .BB1.6

In both conditional expressions, we see inversion of the condition code and the targets, to avoid redundant jump-arounds, as discussed in class and lecture notes #5.

EXAMPLE 4

Input:

```

int f()
{
    int a,b;
    a=g(b+3,5);
}

```

Output:

```

AST Dump for function f:
LIST {
  ASSIGNMENT
    stab_var name=a def @<stdin>:3
  FNCALL, 2 arguments
    stab_fn name=g def @<stdin>:4
  arg #1=
    BINARY OP +
    stab_var name=b def @<stdin>:3
    CONSTANT: (type=int)3
  arg #2=
    CONSTANT: (type=int)5
}

```

```

f:
.BB1.1
    %T00001 = ADD b{lvar},3
    ARG    0,%T00001
    ARG    1,5
    a{lvar} = CALL    $g,2
    RETURN

```

Discussion: I have represented function calls with the CALL quad and introduced the ARG quad to take a value and pass it to the called function. The assembly-language implementation of ARG could vary greatly. On X86-32, it would be a `movl` to the stack at a positive offset from `%esp`, assuming the stack had been pre-adjusted as discussed in lecture notes #6, or a `pushl`, with the arguments pushed in reverse order. On X86-64, it would be a `movq` to the appropriate register (again, wait for unit #6) or the stack if there are more than 6 arguments.

On SPARC the ARG would translate to a MOV to the appropriate `%oX` register for args 0 through 5, but a move to the stack for higher-numbered arguments. Because the argument position might be needed to translate the ARG quad into assembly, I have included it as the first source operand, and the total number of arguments as well in the CALL quad.

Alternately, you could extend the notion of the quad so that one of the operands would be a list of arguments. The example would then look like:

```

.BB1.1
    %T00001 = ADD b{lvar},3
    a{lvar} = CALL    $g, (%T00001,5)

```

Note that the CALL takes the address of the called function. Technically, the target address of the CALL could be any arbitrary expression which yields type "pointer to function". You are only required to handle simple function calls where the target is an identifier of function type.