Announcements

• **Homework 3:** Compiling LLVMlite
• **Goal:**
  – Familiarize yourself with (a subset of) the LLVM IR
  – Implement a translation down to (inefficient) X86lite

• **Available:** later TODAY (tomorrow a.m. at latest)
  – look for a Piazza post

• **Due:** Thursday, Feb. 23rd

• Thursday's lecture will walk through the project in more detail

START EARLY!!
define @factorial(%n) {
    entry:
    %1 = alloca
    %acc = alloca
    store %n, %1
    store 1, %acc
    br label %start

    start:
    %3 = load %1
    %4 = icmp sgt %3, 0
    br %4, label %then, label %else

    then:
    %6 = load %acc
    %7 = load %1
    %8 = mul %6, %7
    store %8, %acc
    %9 = load %1
    %10 = sub %9, 1
    store %10, %1
    br label %start

    else:
    %12 = load %acc
    ret %12

    }

Zdancewic     CIS 341: Compilers
LL Basic Blocks and Control-Flow Graphs

• LLVM enforces (some of) the basic block invariants syntactically.
• Representation in OCaml:

```ocaml
type block = {
    insns : (uid * insn) list;
    terminator : terminator
}
```

• A control flow graph is represented as a list of labeled basic blocks with these invariants:
  – No two blocks have the same label
  – All terminators mention only labels that are defined among the set of basic blocks
  – There is a distinguished, unlabeled, entry block:

```ocaml
type cfg = block * (lbl * block) list
```
LL Storage Model: Locals

• Several kinds of storage:
  – Local variables (or temporaries):  `%uid`
  – Global declarations (e.g. for string constants):  `@gid`
  – Abstract locations: references to (stack-allocated) storage created by the `alloca` instruction
  – Heap-allocated structures created by external calls (e.g. to `malloc`)

• Local variables:
  – Defined by the instructions of the form  `%uid = ...`
  – Must satisfy the `single static assignment` invariant
    • Each `%uid` appears on the left-hand side of an assignment only once in the entire control flow graph.
    • The value of a `%uid` remains unchanged throughout its lifetime
  – Analogous to “let `%uid = e in ...” in OCaml

• Intended to be an abstract version of machine registers.
• We’ll see later how to extend SSA to allow richer use of local variables
  – `phi nodes`
LL Storage Model: \texttt{alloca}

- The \texttt{alloca} instruction allocates stack space and returns a reference to it.
  - The returned reference is stored in local:
    \%ptr = \texttt{alloca typ}
  - The amount of space allocated is determined by the type

- The contents of the slot are accessed via the \texttt{load} and \texttt{store} instructions:

  \begin{verbatim}
  \%acc = \texttt{alloca i64} ; allocate a storage slot
  \texttt{store i64 341, i64* \%acc} ; store the integer value 341
  \%x = \texttt{load i64, i64* \%acc} ; load the value 341 into \%x
  \end{verbatim}

- Gives an abstract version of stack slots
Compiling Structured Data

- Consider C-style structures like those below.
- How do we represent `Point` and `Rect` values?

```c
struct Point { int x; int y; };

struct Rect { struct Point ll, lr, ul, ur };

struct Rect mk_square(struct Point ll, int len) {
    struct Rect square;
    square.ll = square.lr = square.ul = square.ur = ll;
    square.lr.x += len;
    square.ul.y += len;
    square.ur.x += len;
    square.ur.y += len;
    return square;
}
```
Representing Structs

```c
struct Point { int x; int y; };
```

- Store the data using two contiguous words of memory.
- Represent a `Point` value `p` as the address of the first word.

```
p  |   x   |   y   |
```

```c
struct Rect { struct Point ll, lr, ul, ur };
```

- Store the data using 8 contiguous words of memory.

```
square  | ll.x | ll.y | lr.x | lr.y | ul.x | ul.y | ur.x | ur.y |
```

- Compiler needs to know the `size` of the struct at compile time to allocate the needed storage space.
- Compiler needs to know the `shape` of the struct at compile time to index into the structure.
Assembly-level Member Access

Consider: \[[\text{square}.ul.y]\] = (x86.operand, x86.insns)

Assume that \%rcx holds the base address of \text{square}

Calculate the offset relative to the base pointer of the data:
  - ul = sizeof(struct Point) + sizeof(struct Point)
  - y = sizeof(int)

So: \[[\text{square}.ul.y]\] = (ans, Movq 20(\%rcx) ans)
• How to lay out non-homogeneous structured data?

```c
struct Example {
    int x;
    char a;
    char b;
    int y;
};
```
When we do an assignment in C as in:

```c
struct Rect mk_square(struct Point ll, int elen) {
    struct Square res;
    res.lr = ll;
    ...
```

then we copy all of the elements out of the source and put them in the target. Same as doing word-level operations:

```c
struct Rect mk_square(struct Point ll, int elen) {
    struct Square res;
    res.lr.x = ll.x;
    res.lr.y = ll.x;
    ...
```

• For really large copies, the compiler uses something like `memcpy` (which is implemented using a loop in assembly).
C Procedure Calls

• Similarly, when we call a procedure, we copy arguments in, and copy results out.
  – Caller sets aside extra space in its frame to store results that are bigger than will fit in \%rax.
  – We do the same with scalar values such as integers or doubles.
• Sometimes, this is termed "call-by-value".
  – This is bad terminology.
  – Copy-in/copy-out is more accurate.
• Benefit: locality
• Problem: expensive for large records…

• In C: can opt to pass *pointers* to structs: “call-by-reference”

• Languages like Java and OCaml always pass non-word-sized objects by reference.
The caller passes in the address of the point and the address of the result (1 word each).

Note that returning references to stack-allocated data can cause problems.
  – Need to allocate storage in the heap…
ARRAYS
Arrays

void foo() {
    char buf[27];
    buf[0] = 'a';
    buf[1] = 'b';
    // ...
    buf[25] = 'z';
    buf[26] = 0;
}

void foo() {
    char buf[27];
    *(buf) = 'a';
    *(buf+1) = 'b';
    // ...
    *(buf+25) = 'z';
    *(buf+26) = 0;
}

- Space is allocated on the stack for buf.
  - Note, without the ability to allocated stack space dynamically (C’s 
    alloca function) need to know size of buf at compile time...

- buf[i] is really just: (base_of_array) + i * elt_size
Multi-Dimensional Arrays

• In C, \texttt{int M[4][3]} yields an array with 4 rows and 3 columns.
• Laid out in \textit{row-major} order:

\begin{tabular}{cccccccc}
M[0][0] & M[0][1] & M[0][2] & M[1][0] & M[1][1] & M[1][2] & M[2][0] & ... \\
\end{tabular}

• \texttt{M[i][j]} compiles to?

• In Fortran, arrays are laid out in \textit{column major order}.

\begin{tabular}{cccccccc}
M[0][0] & M[1][0] & M[2][0] & M[3][0] & M[0][1] & M[1][1] & M[2][1] & ... \\
\end{tabular}

• In ML and Java, there are no multi-dimensional arrays:
  – (int array) array is represented as an array of pointers to arrays of ints.
• Why is knowing these memory layout strategies important?
Array Bounds Checks

- Safe languages (e.g. Java, C#, ML but not C, C++) check array indices to ensure that they’re in bounds.
  - Compiler generates code to test that the computed offset is legal
- Needs to know the size of the array… where to store it?
  - One answer: Store the size before the array contents.

```
arr

```

- Other possibilities:
  - Pascal: only permit statically known array sizes (very unwieldy in practice)
  - What about multi-dimensional arrays?
Array Bounds Checks (Implementation)

- Example: Assume `%rax` holds the base pointer (arr) and `%ecx` holds the array index `i`. To read a value from the array `arr[i]`:
  
  ```
  movq -8(%rax) %rdx       // load size into rdx
  cmpq %rdx %rcx           // compare index to bound
  j l __ok                 // jump if 0 <= i < size
  callq __err_oob         // test failed, call the error handler
  __ok:
  movq (%rax, %rcx, 8) dest // do the load from the array access
  ```

- Clearly more expensive: adds move, comparison & jump
  - More memory traffic
  - Hardware can improve performance: executing instructions in parallel, branch prediction

- These overheads are particularly bad in an inner loop
- Compiler optimizations can help remove the overhead
  - e.g. In a for loop, if bound on index is known, only do the test once
C-style Strings

• A string constant "foo" is represented as global data:
  _string42: 102 111 111 0

• C uses null-terminated strings
• Strings are usually placed in the text segment so they are read only.
  – allows all copies of the same string to be shared.

• Rookie mistake (in C): write to a string constant.

  ```c
  char *p = "foo";
  p[0] = 'b';
  ```

• Instead, must allocate space on the heap:

  ```c
  char *p = (char *)malloc(4 * sizeof(char));
  strncpy(p, "foo", 4);  /* include the null byte */
  p[0] = 'b';
  ```
DATATYPES IN THE LLVM IR
Structured Data in LLVM

- LLVM’s IR uses types to describe the structure of data.

\[
\begin{align*}
\text{t ::=} & \quad \text{void} \\
& \quad i1 \mid i8 \mid i64 \quad \text{N-bit integers} \\
& \quad [\langle \#\text{elts} \rangle \times t] \quad \text{arrays} \\
\text{fty} & \quad \{t_1, t_2, \ldots, t_n\} \quad \text{function types} \\
\text{t*} & \quad \text{structures} \\
\text{%Tident} & \quad \text{pointers} \\
\text{fty ::= Function Types} & \quad \text{named (identified) type} \\
& \quad t (t_1, \ldots, t_n) \quad \text{return, argument types}
\end{align*}
\]

- \langle \#\text{elts} \rangle \text{ is an integer constant } \geq 0
- Structure types can be named at the top level:

\[
\text{%T1 = type } \{t_1, t_2, \ldots, t_n\}
\]

- Such structure types can be recursive
Example LL Types

• An array of 341 integers: \([ 341 \times \text{i64} ]\)

• A two-dimensional array of integers: \([ 3 \times [ 4 \times \text{i64} ] ]\)

• Structure for representing arrays with their length:
  \(\{ \text{i64}, [0 \times \text{i64}] \}\)
  – There is no array-bounds check; the static type information is only used for calculating pointer offsets.

• C-style linked lists (declared at the top level):
  \(%\text{Node} = \text{type} \{ \text{i64}, \%\text{Node}* \}\)

• Structs from the C program shown earlier:
  \(%\text{Rect} = \{ \%\text{Point}, \%\text{Point}, \%\text{Point}, \%\text{Point} \}\)
  \(%\text{Point} = \{ \text{i64}, \text{i64} \}\)
• LLVM provides the `getelementptr` instruction to compute pointer values
  - Given a pointer and a “path” through the structured data pointed to by that pointer, `getelementptr` computes an address
  - This is the abstract analog of the X86 LEA (load effective address). It does not access memory.
  - It is a “type indexed” operation, since the sizes computations involved depend on the type

```
insn ::= ...
    | `getelementptr` t* %val, t1 idx1, t2 idx2 ,...
```

• Example: access the x component of the first point of a rectangle:

```
%tmp1 = `getelementptr` %Rect* %square, i32 0, i32 0
%tmp2 = `getelementptr` %Point* %tmp1, i32 0, i32 0
```
```c
struct RT {
    int A;
    int B[10][20];
    int C;
};
struct ST {
    struct RT X;
    int Y;
    struct RT Z;
};
int *foo(struct ST *s) {
    return &s[1].Z.B[5][13];
}
```

%RT = type { i32, [10 x [20 x i32]], i32 }
%ST = type { %RT, i32, %RT }
define i32* @foo(%ST* %s) {
    entry:
        %arrayidx = getelementptr %ST* %s, i32 1, i32 2, i32 1, i32 5, i32 13
        ret i32* %arrayidx
}

Final answer: ADDR + size_ty(%ST) + size_ty(%RT) + size_ty(i32)
+ size_ty(i32) + 5*20*size_ty(i32) + 13*size_ty(i32)

*adapted from the LLVM documentation: see http://llvm.org/docs/LangRef.html#getelementptr-instruction
• GEP *never* dereferences the address it’s calculating:
  – GEP only produces pointers by doing arithmetic
  – It doesn’t actually traverse the links of a datastructure

• To index into a deeply nested structure, need to “follow the pointer” by loading from the computed pointer
  – See list.ll from HW3
Compiling Datastructures via LLVM

1. Translate high level language types into an LLVM representation type.
   - For some languages (e.g. C) this process is straightforward
     • The translation simply uses platform-specific alignment and padding
   - For other languages, (e.g. OO languages) there might be a fairly complex elaboration.
     • e.g. for Ocaml, arrays types might be translated to pointers to length-indexed structs.

\[
\text{[int array]} = \{ \text{i32}, [0 \times \text{i32}] \}^* 
\]

2. Translate accesses of the data into getelementptr operations:
   - e.g. for Ocaml array size access:
     \[
     \llbracket \text{length a} \rrbracket = \\
     \%1 = \text{getelementptr} \{ \text{i32}, [0\times \text{i32}] \}^* \%a, \text{i32 0, i32 0} 
     \]
Bitcast

- What if the LLVM IR’s type system isn’t expressive enough?
  - e.g. if the source language has subtyping, perhaps due to inheritance
  - e.g. if the source language has polymorphic/generic types

- LLVM IR provides a **bitcast** instruction
  - This is a form of (potentially) unsafe cast. Misuse can cause serious bugs (segmentation faults, or silent memory corruption)

```llvm
%rect2 = type { i64, i64 } ; two-field record
%rect3 = type { i64, i64, i64 } ; three-field record

define @foo() {
  %1 = alloca %rect3 ; allocate a three-field record
  %2 = bitcast %rect3* %1 to %rect2* ; safe cast
  %3 = getelementptr %rect2* %2, i32 0, i32 1 ; allowed
  ...
}
```
see HW3

LLVMLITE SPECIFICATION
Discussion: Defining a Language

• Premise: programming languages are purely ‘formal’ objects
  – We (as language designers) get to determine the meaning of the language constructs

• Question: How do we specify that meaning?

• Question: What are the properties of a good specification?

• Examples?
Approaches to Language Specification

- Implementation
  - It does what it does!

- Social
  - Authority figure says: “it means X”
  - English prose

- Technological
  - Multiple implementations
  - Reference interpreter
  - Test cases / Examples

- Translation
  - Semantics given in terms of (hopefully better specified) target

- Mathematical
  - “Informal” specifications
  - “Formal” specifications

Less “formal”: Techniques may miss problems in programs

This isn’t a tradeoff... all of these methods should be used.

Even the most “formal” can still have holes:
- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth. “Beware of bugs in the above code; I have only proved it correct, not tried it.”

More “formal”: eliminate with certainty as many problems as possible.

CIS 500: Fall 2014
LLVMLite notes

• Reall LLVM requires that constants appearing in getelementptr be declared with type i32:

```
%struct = type { i64, [5 x i64], i64}
@gbl = global %struct {i64 1, 
    [5 x i64] [i64 2, i64 3, i64 4, i64 5, i64 6], i64 7}

define void @foo() {
    %1 = getelementptr %struct* @gbl, i32 0, i32 0
    ...
}
```

• LLVMLite ignores the i32 annotation and treats these as i64 values
  – we keep the i32 annotation in the syntax to retain compatibility with the clang compiler
COMPILING LLVM-LITE TO X86
Compiling LLVMlite Types to X86

- $[i1], [i64], [t^*] =$ quad word (8 bytes, 8-byte aligned)
- raw $i8$ values are not allowed (they must be manipulated via $i8^*$)
- array and struct types are laid out sequentially in memory

- getelementptr computations must be relative to the LLVMlite size definitions
  - i.e. $[i1] = $quad
Compiling LLVM locals

• How do we manage storage for each %uid defined by an LLVM instruction?

• Option 1:
  – Map each %uid to a x86 register
  – Efficient!
  – Difficult to do effectively: many %uid values, only 16 registers

• Option 2:
  – Map each %uid to a stack-allocated space
  – Less efficient!
  – Simple to implement

• For HW3 we will follow Option 2
Other LLVMlite Features

• Globals
  – must use %rip relative addressing

• Calls
  – Follow x64 AMD ABI calling conventions
  – Should interoperate with C programs

• getelementptr
  – trickiest part
see HW3 and README

ll.ml, using main.native, clang, etc.

TOUR OF HW 3
TAGGED DATATYPES
C-style Enumerations / ML-style datatypes

• In C:
  ```c
  enum Day {sun, mon, tue, wed, thu, fri, sat} today;
  ```

• In ML:
  ```ml
  type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  ```

• Associate an integer *tag* with each case: *sun = 0, mon = 1, …*
  - C lets programmers choose the tags

• ML datatypes can also carry data:
  ```ml
  type foo = Bar of int | Baz of int * foo
  ```

• Representation: a *foo* value is a pointer to a pair: (*tag, data*)

• Example: *tag(Bar) = 0, tag(Baz) = 1* 
  ```ml
  let f = Bar(3) = [let f = Bar(3)] = (0, 3)

  let g = Baz(4, f) = [let g = Baz(4, f)] = (1, 4, f)
  ```
Switch Compilation

• Consider the C statement:

```c
switch (e) {
    case sun: s1; break;
    case mon: s2; break;
    ...
    case sat: s3; break;
}
```

• How to compile this?
  – What happens if some of the break statements are omitted? (Control falls through to the next branch.)
Cascading if s and Jumps

\[
\text{[switch(e) \{case tag1: s1; case tag2 s2; \ldots\}] =}
\]

- Each \$tag1\ldots$tagN is just a constant int tag value.

- Note: \[break;\] (within the switch branches) is:
  \[br \%merge\]

\[
\%tag = [e];
br label \%l1
l1: \%cmp1 = icmp eq \%tag, \$tag1
    br \%cmp1 label \%b1, label \%merge
b1: \[s1]\n    br label \%l2

l2: \%cmp2 = icmp eq \%tag, \$tag2
    br \%cmp2 label \%b2, label \%merge
b2: \[s2]\n    br label \%l3
...

lN: \%cmpN = icmp eq \%tag, \$tagN
    br %cmpN label %bN, label %merge
bN: \[sN]\n    br label %merge
merge:
\]
Alternatives for Switch Compilation

• Nested if-then-else works OK in practice if # of branches is small
  – (e.g. < 16 or so).

• For more branches, use better datastructures to organize the jumps:
  – Create a table of pairs \( (v1, \text{branch\_label}) \) and loop through
  – Or, do binary search rather than linear search
  – Or, use a hash table rather than binary search

• One common case: the tags are dense in some range \([\text{min}...\text{max}]\)
  – Let \( N = \text{max} - \text{min} \)
  – Create a branch table \( \text{Branches}[N] \) where \( \text{Branches}[i] = \text{branch\_label} \) for tag \( i \).
  – Compute \( \text{tag} = \llbracket e \rrbracket \) and then do an *indirect jump*: \( J \text{Branches}[\text{tag}] \)

• Common to use heuristics to combine these techniques.
**ML-style Pattern Matching**

- ML-style match statements are like C’s switch statements except:
  - Patterns can bind variables
  - Patterns can nest

- Compilation strategy:
  - “Flatten” nested patterns into matches against one constructor at a time.
  - Compile the match against the tags of the datatype as for C-style switches.
  - Code for each branch additionally must copy data from \([e]\) to the variables bound in the patterns.

- There are many opportunities for optimization, many papers about “pattern-match compilation”
  - Many of these transformations can be done at the AST level