CS 473: COMPILER DESIGN
INTERMEDIATE REPRESENTATIONS
Compilation in a Nutshell

Source Code
(Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0 ; }

Abstract Syntax Tree:
If
  Eq
  Assn
    b
    0
    a
    1

Analysis & Transformation

Lexical Analysis

Parsing

Backend

Assembly Code
11:
cmpq %eax, $0
ejq 12
jmp 13
12:...

...
Compilation in a Nutshell

**Abstract Syntax Tree:**

```
If
  Eq
  Assn
  None
```

**Intermediate code:**

```
l1:
  %cnd = icmp eq i64 %b, 0
  br i1 %cnd, label %l2, label %l3
l2:
  store i64* %a, 1
  br label %l3
l3:
```

**Assembly Code**

```
l1:
cmpq %eax, $0
jeq l2
jmp l3
l2:
...```
Intermediate Representations (IRs)

- Abstract machine code: hides details of the target architecture
- Allows machine independent code generation and optimization
Intermediate Representations (IRs)

- Abstract machine code: hides details of the target architecture
- Allows machine independent code generation and optimization
Multiple IRs

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
- In practice, multiple intermediate representations might be used (for different purposes)
Exercise: What are some language constructs that exist in source code but don’t exist in assembly?

High-level IRs
- Abstract syntax + extra information, like types
- Structured control flow, variable names, methods, functions, etc.
- Allows high-level optimizations based on program structure
  - e.g. inlining “small” functions, reuse of constants, etc.
- Useful for semantic analyses like type checking

Low-level IRs
- Machine dependent assembly code + extra pseudo-instructions (malloc, etc.)
- Source structure of the program is lost (to get closer to assembly structure)
- Allows low-level optimizations based on target architecture
  - e.g. register allocation, instruction selection, memory layout, etc.

An IR will be somewhere between high-level and low-level for each feature
What makes a good IR?

• An IR can be anything we want!

• Easy translation target (from the level above)
• Easy to translate (to the level below)
• Easy to analyze and transform (at the current level)
  – Fewer constructs means simpler phases/optimizations

• Example: Source language might have arrays, structs, and unions, while target just has addresses in memory
• Translation from source to IR might map arrays, structs, etc. into a single underlying composite data structure
Questions

Top
Intermediate Representations (IRs)

- Abstract machine code: hides details of the target architecture
- Allows machine independent code generation and optimization
## Tree IR: Overview

<table>
<thead>
<tr>
<th>Source Language</th>
<th>Tree IR</th>
<th>Target Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>A program is a series of statements</td>
<td>A program is a series of statements</td>
<td>A program is a series of statements</td>
</tr>
<tr>
<td>Variables must be declared, have types</td>
<td>Variables aren’t declared, don’t have types</td>
<td>No variables, just a fixed set of registers</td>
</tr>
<tr>
<td>Arrays, but no pointers – don’t explicitly reference memory</td>
<td>Pointers, no arrays – calculate offsets in memory</td>
<td>Pointers, no arrays – calculate offsets in memory</td>
</tr>
<tr>
<td>Structured control flow: conditionals, loops</td>
<td>Unstructured control flow: labels, jumps</td>
<td>Unstructured control flow: labels, jumps</td>
</tr>
<tr>
<td>Expressions can be deeply nested and complex</td>
<td>Expressions can be deeply nested and complex, might even access memory!</td>
<td>One operation (addition, subtraction, comparison, memory access) per expression</td>
</tr>
<tr>
<td>Written by users</td>
<td>Nobody sees it but the compiler!</td>
<td>Run by the CPU</td>
</tr>
</tbody>
</table>
IR Translation

• Goal: convert AST into IR tree (same basic structure, simpler ops)
• Approach: write a recursive function over ASTs that outputs the IR node corresponding to each AST node
  – Just like analysis, but instead of recording data or generating error messages, we’re building a new tree!
  – In each case, build the IR tree for that kind of node
• For each node type that doesn’t exist in the IR (variables, loops, etc.), build a more complex subtree that simulates it

```
If
  b
  Assign
    Subscr
      a
      i
    0

Cjmp
  b
  Move
    Mem
      0
  Label2

Plus
  fp
  1
```
# Tree IR: Overview

<table>
<thead>
<tr>
<th>Source Language</th>
<th>Tree IR</th>
<th>Target Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>A program is a series of statements</td>
<td>A program is a series of statements</td>
<td>A program is a series of statements</td>
</tr>
<tr>
<td>Variables must be declared, have types</td>
<td>Variables aren’t declared, don’t have types</td>
<td>No variables, just a fixed set of registers</td>
</tr>
<tr>
<td>Arrays, but no pointers – don’t explicitly reference memory</td>
<td>Pointers, no arrays – calculate offsets in memory</td>
<td>Pointers, no arrays – calculate offsets in memory</td>
</tr>
<tr>
<td>Structured control flow: conditionals, loops</td>
<td>Unstructured control flow: labels, jumps</td>
<td>Unstructured control flow: labels, jumps</td>
</tr>
<tr>
<td>Expressions can be deeply nested and complex</td>
<td>Expressions can be deeply nested and complex, might even access memory!</td>
<td>One operation (addition, subtraction, comparison, memory access) per expression</td>
</tr>
</tbody>
</table>
## Tree IR: Variables

### Source

```c
int a = 4;
int result = 5;
```

### Target

```assembly
move %eax, 4
move %ebx, 5
```

- Unlimited number of variables
- Can declare more at various points
- Locals go in and out of scope
- All variables accessed in the same way
- Fixed number of registers, no way to make more
- Extra variables need to be stored somewhere else!
- Variables don’t go away until overwritten
Tree IR: Variables

Source

```plaintext
int a = 4;
int result = 5;
```

Target

```plaintext
move %eax, 4
move %ebx, 5
```

- Unlimited number of variables
- Can declare more at various points
- Locals go in and out of scope
- All variables accessed in the same way

- Fixed number of registers, no way to make more
- Extra variables need to be stored somewhere else!
  - In memory: on the stack
- Variables don’t go away until overwritten
int a = 4;
int b = 5; // {a : int, b : int}
int f(int c, int d){ // {a : int, b : int, c : int, d : int}
    int e; // {a : int, b : int, c : int, d : int, e : int}
    ...
} // {a : int, b : int}

f(1, 2);

a = 4   b = 5
int a = 4;
int b = 5;  // {a : int, b : int}
int f(int c, int d){  // {a : int, b : int, c : int, d : int}
    int e;  // {a : int, b : int, c : int, d : int, e : int}
    ...
}  // {a : int, b : int}

f(1, 2);

\[
\begin{array}{c}
e = ? \\
c = 1 \quad d = 2 \\
a = 4 \quad b = 5
\end{array}
\]
int a = 4;
int b = 5;    // {a : int, b : int}
int f(int c, int d){  // {a : int, b : int, c : int, d : int}
    int e;        // {a : int, b : int, c : int, d : int, e : int}
    ...
}
      // {a : int, b : int}

f(1, 2);

```
e = ?
c = 1   d = 2
a = 4   b = 5
```
• Stack frames are created in the program when it runs, to store (some) local variables and function arguments

• Push when variables come into scope (new blocks/declarations, function calls), pop when they go out of scope

• Need to exist when code runs, but source code just assumes they happen
  – So the translated code needs to explicitly create them!

• Goal: implement a stack frame data structure in the IR/target language, and insert code that uses it when we translate
Questions
Anatomy of a Stack Frame

int a = 4;
int b = 5;
int f(int c, int d){
    int e = 6;
    ...
}

Exercise: what information do we need to store in a stack frame for f?
f(1, 2);

c = 1  d = 2  e = 6

arguments  return address  local variables
Making a Stack Frame

f(1, 2);

push 1
push 2
push ret_addr
call f
ret_addr:
pop
...

c = 1  d = 2

Arguments  Return Address  Local Variables

e = 6
Using a Stack Frame

```c
int f(int c, int d){
    int e = 6;
    e = c + d;
}
```

```
f:
stack.var[0] = 6
jmp stack.ret
```
Using a Stack Frame

```c
int f(int c, int d){
    int e = 6;
    e = c + d;
}
```

**f:** *(fp + 3) = 6
don't modify c or d


jmp stack.ret

- *fp*
- \(c = 1\)
- \(d = 2\)
- \(e = 6\)
- arguments
- return address
- local variables
Using a Stack Frame

```c
int f(int c, int d){
    int e = 6;
    e = c + d;
}
```

- **f:**
  - \(*\(fp + 3\) = 6\)
  - \(*\(fp + 3\) = *\(fp + 0\) + *\(fp + 1\)\)
  - jmp stack.ret

```
c = 1   d = 2   e = 6
```

- **fp**
  - arguments
  - return address
  - local variables
Stack frames are created in the program when it runs, to store (extra) variables.

Push when variables come into scope (new blocks/declarations, function calls), pop when they go out of scope.

Need to exist when code runs, but aren’t in source code—So the translated code needs to explicitly create them!

We implement a stack frame data structure in the IR/target language, and insert code that uses it when we translate—All references to stack-allocated variables turn into memory accesses to offsets from the frame pointer.
Questions

Top
Course Project

• Posted on the course website
  (https://www.cs.uic.edu/~mansky/teaching/cs473/sp21/project.html)

• Choose one of the assignments and expand on it, or add a (small) language feature and change all the relevant stages of the compiler
  – Examples: add records to the language (lexer, parser, type checker, IR translation), or add classes and objects to just the parser
  – More examples on the web page

• Work individually or in a group of 2

• Initial project proposal due Monday 3/15 at the start of class
• Project should be slightly larger than a single assignment – if yours sounds too big or too small, we’ll discuss and figure something out
• Submit via Gradescope
• If you’re not sure what to propose, stop by office hours and we can talk about it!
int f()
{
    int a = 5;
    int b = 6;
    ...b...
}

Tree IR: Variables
• We might not need to store a variable on the stack: assembly languages have a few *registers* for storing variables
• This alternative is definitely not possible if we ever need a pointer to the variable (e.g., passed by reference)
• We can’t always tell this when it’s declared, so we might need to change it later (register allocation)
Questions

Top
Tree IR: What’s in the IR?

- demo: ast.h vs. tree.h

Differences between AST and IR:

- IR has memory addresses (T_MEM) instead of arrays, strings, etc.
- An IR exp might need to do a sequence of statements first (T_ESEQ)
- IR has no declarations: variables don’t have types and don’t need to be declared, initializers translate to just normal assignments (T_MOVE)
- IR has labels and jumps (T_JUMP, T_CJUMP) instead of if and while statements
Tree IR: Arrays

- $\lbrack a \rbrack$ means “the result of translating $a$” (in code, a recursive call to the translation function)
• \([a]\) means “the result of translating \(a\)” (in code, a recursive call to the translation function)
Tree IR: Arrays

- \([a]\) means “the result of translating \(a\)” (in code, a recursive call to the translation function)
- Exercise: Using the constructors from tree.h, build the IR tree on the right-hand side of the arrow
Program 4: IR Translation

• Posted on the course website (https://www.cs.uic.edu/~mansky/teaching/cs473/sp21/program4.html)

• Fill in pieces of the IR translation

• First submission due next Wednesday at the start of class

• Submit via Gradescope

• Writing a translation function on ASTs is just like typechecking:
  1. One function for each AST node type (expressions, statements, etc.)
  2. In each function, a switch statement over the kinds of nodes for that nonterminal
  3. In each case, use tree IR constructors to build the corresponding subtree, and make recursive calls for any child nodes
Questions
Tree IR: Array Bounds

```
Subscr
  a
  i
```

```
 addr0 <a[0]>
 addr1 <a[1]>
 addr2 5
```

```
Mem
  Plus
    [a] [i]
```

```
a[2]
```

```
*(addr0 + 2)
```
Tree IR: Array Bounds

```
Subscr
  a
  i
```

```
addr0  2
addr1  <a[0]>
addr2  <a[1]>
addr3  5
```

```
Mem
  Plus
    [a]
    [i]
    1
```

```
Plus
  a
  i
```

```
addr0  2
addr1  <a[0]>
addr2  <a[1]>
addr3  5
```
if i < Mem([a]) then

else jmp (out-of-bounds error)
Tree IR: Conditionals

If

\[ \begin{align*}
  c & \quad a & \quad b
\end{align*} \]

Cjmp

\[ \begin{align*}
  [c] & \quad \text{labela} & \quad \text{labelb}
\end{align*} \]

\[ \begin{align*}
  \text{labela} & \quad [a] & \quad \text{labelb} & \quad [b]
\end{align*} \]
Tree IR: Conditionals

Cjmp \([c]\) labela labelb
labela:
[a]
labelb:
[b]
If the LHS starts with `Mem`, this is a store to memory
If it starts with `Temp`, this is an assignment to a temporary register
Exercise: draw the IR tree for the translation of
\[ a[3] = b[1]; \]
where \( a \) and \( b \) are local (Temp node) variables.
Exercise: draw the IR tree for the translation of
\[ a[3] = b[1]; \]
where \(a\) and \(b\) are local (Temp node) variables
Questions
Tree IR: String Literals

- Keep a list of “fragments” on the side that need to be translated separately from the program.
While
  a
  b

While
  true
  b

label-start

b

Jmp

label-start
Tree IR: Loops

```
start:
Cjmp [a]
body done

body:
[b]
Jmp start
done:
```

```
start:
[b]
Jmp start
done:
```
Questions

Top
The IR’s Call is really just a fancy jump! (that remembers the values of its arguments)

We’ll manage the stack frame when translating the function body
Tree IR: Function Calls

- **Builtin or external calls**: `print`, string library, memory management, ...
- Can’t be implemented in the source language!
- Must use the calling convention specified by the library/target OS/processor
Questions

Top
• But also: decide whether \( a \) is going to be stored in a register or a stack, and in the latter case, record that we’ll need room for it in the frame

• In code: `Tr_access acc = Tr_allocLocal(level, TRUE); record_var(s->data.vardec_ops.name, acc);`
Tree IR: Declarations

- Also compute the size and layout of the stack frame for this function
  - We use this for `<framesize>`, but also for translating variables in `body`
Questions

Top
Tree IR: Summary

- Recursively translate each AST node into a piece of an IR tree
  - A step closer to target language (assembly code)
- Variables -> temporary registers or stack frame accesses
- Arrays, strings -> memory accesses
- Control flow -> jumps to labels
- Function bodies -> bodies + stack frame setup/teardown
- Can try to make things a little more efficient, but it doesn’t matter too much if we don’t – we’ll optimize later on