CS 473: COMPILER DESIGN
Program 3 Questions

Top
INTERMEDIATE REPRESENTATIONS
Compilation in a Nutshell

Abstract Syntax Tree:

```
If
  Eq
  Assn
  None
b
0
a
1
```

Intermediate code:

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

Assembly Code

```
11:  cmpq %eax, $0
     jeq 12
     jmp 13
12:  ...
```
Intermediate Representations (IRs)

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

Diagram:
- AST -> IR
- IR -> Optimization
- IR -> x86
- IR -> Java Byte-code
- IR -> ARM
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)
What makes a good IR?

• Easy translation target (from the level above)
• Easy to translate (to the level below)
• Narrow interface
  – Fewer constructs means simpler phases/optimizations

• Example: Source language might have “while”, “for”, and “foreach” loops (and maybe more variants)
  – IR might have only “while” loops and sequencing
  – Translation eliminates “for” and “foreach”

\[
\text{⟦for}(\text{pre}; \text{cond}; \text{post}) \ {\text{body}}\text{⟧} = \\
\text{⟦pre}; \text{while}(\text{cond}) \ {\text{body};post}\text{⟧}
\]

– Here the notation \(\text{⟦cmd⟧}\) denotes the “translation” or “compilation” of the command \(\text{cmd}\).
IRs at the extreme

• High-level IRs
  – Abstract syntax + extra information, like types or disambiguated syntax
  – Typically preserves the high-level language constructs
    • Structured control flow, variable names, methods, functions, etc.
    • May do some simplification (e.g. convert `for` to `while`)
  – 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
    • e.g. a pseudo instruction for interfacing with garbage collector or memory allocator
      (parts of the language runtime system)
  – 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.

• What’s in between?
Mid-level IRs: Many Varieties

- Intermediate between AST and assembly
- May have unstructured jumps, abstract registers or memory locations
- Expressions/operations are smaller than in source language, bigger than in target language

- Many examples:
  - Triples: OP a b
  - Quadruples: a = b OP c ("three address form")
  - SSA: variant of quadruples where each variable is assigned exactly once
    - Easy dataflow analysis for optimization
    - e.g. LLVM IR
  - Stack-based:
    - Easy to generate
    - e.g. JVM Bytecode
Questions

Top
Midterm 1 Questions

Top
What Can Regular Expressions Do?

• “Regular expressions can’t count/don’t have memory”

• But they can:

  count to 3
  bbb

  count to 0 mod 2 (or 3, or ...)
  (bb)*

  start and end with the same character
  a(a|b)*a | b(a|b)*b

• So what can’t they do?
  – match parens, maintain ratios (as many a’s as b’s), ...

• They can’t count *arbitrarily high*/have *unbounded* memory
Example 1: Tree-Based IR

- Goal: AST where each node is closer to a machine operation
  - Jumps instead of loops, memory instead of arrays or records
- Expressions: constants, variables, operators, memory addresses, calls
- Statements: assignments, labels, jumps, conditional jumps
- Do less in one expression/statement than in source
- Explicitly manages stack frames for calls

if b then a[i] := 0

```
if b then a[i] := 0
```

```
If
  b
  Assign
    Subscr
      a
      i
    0

Cjmp
  b
  Move
    Mem
      0
    Label2
  Cjmp
    Mem
      0
    Label2
  Plus
    fp
    1
```
int f() {
    int a = 5;
    int b = 6;
    ...b...
}
int f(){
    int b = 6;
    int g(int x, f_stack){
        ...b...
    }
    ...
    g(5, f.stackframe);
}

Mem
  Plus
    fp
    Plus
      f.st
      1

b = 6
int f()
{
    int b = 6;
    int g(int x, f_stack)
    {
        ...b...
    }
    ...g(5, f.stackframe);
}

b = 6
• 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
• $[a]$ means “the result of translating $a$”
Tree IR: Arrays and Records

- \([a]\) means “the result of translating a”
• \([a]\) means “the result of translating a”
• \([a]\) means “the result of translating \(a\)”
• In general, the LHS will translate to a memory access
Tree IR: Arrays and Records

a.id {a : {name : string, id : int}, ...}

offset(id) = 1

- \([a]\) means “the result of translating a”
- In general, the LHS will translate to a memory access
Questions

Top
Tree IR: Array Bounds

Subscr

\( a \) \( i \)

\[ \text{addr0} < a[0] > \]
\[ \text{addr1} < a[1] > \]
\[ \text{addr2} 5 \]

\[ a[2] \]

\[ \text{Mem} \]
\[ \text{Plus} \]
\[ [a] [i] \]

\[ \text{[addr0 + 2]} \]
Tree IR: Array Bounds

Subscr

a

i

Mem

Plus

[a]

Plus

[i]

1

<table>
<thead>
<tr>
<th>addr0</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>addr1</td>
<td>a[1]</td>
</tr>
<tr>
<td>addr2</td>
<td>a[2]</td>
</tr>
<tr>
<td>Addr3</td>
<td>5</td>
</tr>
</tbody>
</table>
if i < Mem(a) then
else

jump (out-of-bounds error)
If
cab

Cjmp
[c]
labela
labelb

Seq
labela
[a]

Seq
labelb
[b]
Tree IR: Conditionals

If
  c
  a
  b

Cjmp [c] labela labelb
labela:
  [a]
labelb:
  [b]

Seq
  Cjmp
    [c]
    labela
    labelb
Seq
  labela
  [a]
Seq
  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
- $[a]$ means “the result of translating $a$”
- In general, the LHS will translate to a memory access

```
Tree IR: Arrays and Records

- Subscr
  - a
  - i
- Field
  - a
  - f

- a means “the result of translating a”
- In general, the LHS will translate to a memory access
```
• Keep a list of “fragments” on the side that need to be translated separately from the program
Tree IR: Loops

While
  a
  b

While
  true
  b

Seq
  label-start
  b
  Jmp
  label-start
Tree IR: Loops

start:
Cjmp [a] body done
body:
[b]
Jmp start
done:

start:
[b]
Jmp start
done:
Tree IR: Loops

**While**

- **start:**
  - Cjmp \([a]\) body done
- **body:**
  - \([b]\)
  - Jmp start
- **done:**

**For**

- **[let i := a in while i < b do (c; i := i + 1)]**
Questions

Top
a[3] := b.x

where

b : {x : int, y : int}
Top
In parsing Tiger, `a & b` is translated into `if a then b else 0`
Tree IR: Conditionals, Revisited

- Since Boolean expressions will almost always be used as conditions, we can translate them as conditions instead of expressions:

- A Boolean expression translates to a conditional jump *with holes*
Since Boolean expressions will almost always be used as conditions, we can translate them as conditions instead of expressions:

A Boolean expression translates to a conditional jump with holes

...
Since Boolean expressions will almost always be used as conditions, we can translate them as conditions instead of expressions:

A Boolean expression translates to a conditional jump with holes

A Boolean expression translates to a conditional jump with holes
Since Boolean expressions will almost always be used as conditions, we can translate them as conditions instead of expressions:

A Boolean expression translates to a conditional jump with holes

This gives us a cleaner representation of complex conditionals, without making simple Boolean assignments worse (Cx in the textbook)
Questions
int f()
{
    int b = 6;
    int g(int x, f_stack){
        ...b...
    }
    ...g(5, f.stackframe);
}

int f()
{
    int b = 6;
    int g(int x, f_stack){
        ...b...
    }
    ...g(5, f.stackframe);
}
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

- **Built-in or external calls**: `print`, string library, memory management, ...
- Can’t be implemented in the source language!
Tree IR: Function Calls

- **Built-in or external calls**: `print`, string library, memory management, ...
- Can’t be implemented in the source language!
Tree IR: Function Calls

- **Builtin or external calls**: `print`, string library, memory management, ...
- Can’t be implemented in the source language!

- Store arguments in registers;
- Store return address on stack;
- Jump to label `_builtin`;
- Move stack pointer back
Tree IR: Function Calls

- *Built-in or external* calls: `print`, string library, memory management, ...
- Can’t be implemented in the source language!
- Each target OS/external library/target language might have its own calling convention, so it’s better to put a stub here and fill this in later.
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.

Type information is already recorded in the type environment/symbol table.
FunctionDec
f args body

f:
MOVE sp, sp + <framesize>
copy args into stack/registers
[body]
MOVE ret <body’s return value>
MOVE sp, sp - <framesize>
JMP <return address>

• 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
Tree IR: Demo
type rec := \{id : int, size : int\}

... 
a := rec \{id = 123, size = 456\}

r = malloc(sizeof(rec));

r->id = 123;
r->size = 456;
return r;
type rec := {id : int, size : int}
...
a := rec {id = 123, size = 456}

r = malloc(2 * wordsize)
r->id = 123;
r->size = 456;
return r;
type rec := \{id : int, size : int\}

...  

a := rec \{id = 123, size = 456\}

r = malloc(2 * wordsize)
Move([r.id], 123)
Move([r.size], 456)
return r;
type rec := {id : int, size : int}
...
a := rec {id = 123, size = 456}

r = malloc(2 * wordsize)
Move([r.id], 123)
Move([r.size], 456)
Temp(r)
Tree IR: Arrays and Records

type rec := {id : int, size : int}
...

a := rec {id = 123, size = 456}

\[ r = \text{malloc}(2 \times \text{wordsize}) \]
\[ \text{Move}([r.id], 123) \]
\[ \text{Move}([r.size], 456) \]
\[ \text{Temp}(r) \]
type arr := array of int
...
type arr := array of int
...
e := ...
...
a := arr[e] of 5

- We might not know statically how big the array should be!
- Option 1: construct a loop

\[
arr[e_1] \text{ of } e_2 \quad \rightarrow \quad \begin{array}{l}
\text{r} = \text{malloc(...)}; \\
\text{for i from 0 to } e_1 \text{ do } r[i] = e_2
\end{array}
\]

- Option 2: outsource to an external function

\[
arr[e_1] \text{ of } e_2 \quad \rightarrow \quad \begin{array}{l}
\text{initArray}(e_1, e_2)
\end{array}
\]
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, records, 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