CS 536 Announcements for Wednesday, April 10, 2024

Last Time
- variable access at runtime
  - local vs global variables
  - static vs dynamic scopes

Today
- wrap up variable access at runtime
- start looking at details of MIPS
- code generation

Next Time
- continue code generation

Dynamic non-local scope

Example

```plaintext
function main() {
    int a = 0;
    fun1();
    fun2();
}
function fun2() {
    int a = 27;
    fun1();
}
function fun1() { a = a + 1; }
```

Key point – we don’t know which non-local variable we are referring to

Two ways to set up dynamic access
- deep access – somewhat similar to access links
- shallow access – somewhat similar to displays

Deep access
- if the variable isn’t local
  - follow control link to caller’s AR
  - check to see if it defines the variable
  - if not, follow the next control link down the stack
- note that we need to know if a variable is defined with that name in an AR
  - usually means we’ll have to associate a name with a stack slot

Shallow access
- keep a table with an entry for each variable declaration
- compile a direct reference to that entry
- at function call on entry to function F
  - F saves (in its AR) the current values of all variables that F declares itself
  - F restores these values when it finishes
Compiler Big Picture

Scanner

Parser

Static-Semantic Analysis

IR Codegen

Optimizer

MC Codegen
Compiler Back End: Design Decisions

When do we generate?
- directly from AST
- during SDT

How many passes?
- fewer passes
- more passes

What do we generate?
- machine code
- intermediate representation (IR)

Possible IRs
- CFG (control-flow graph)
- 3AC (three-address code)
  - instruction set for a fictional machine
  - every operator has at most 3 operands
  - provides illusion of infinitely many registers
  - "flatten out" expressions
3AC Example

3AC instruction set

**Assignment**
- \( x = y \text{ op } z \)
- \( x = \text{ op } y \)
- \( x = y \)

**Indirection**
- \( x = y[z] \)
- \( y[z] = x \)
- \( x = &y \)
- \( x = *y \)
- \( *y = x \)

**Call/Return**
- \( \text{param } x, k \)
- \( \text{retval } x \)
- \( \text{call } p \)
- \( \text{enter } p \)
- \( \text{leave } p \)
- \( \text{return} \)
- \( \text{retrieve } x \)

**Type Conversion**
- \( x = \text{AtoB } y \)

**Jumps**
- \( \text{if ( x op y) goto } L \)

**Labeling**
- \( \text{label } L \)

**Basic Math**
- \( \text{times, plus, etc.} \)

---

Example

**source code**

```plaintext
if \( x + y * z > x * y + z \) [
  a = 0.
]

b = 2.
```

**3AC code**

```plaintext
tmp1 = y * z

tmp2 = x + tmp1

tmp3 = x * y

tmp4 = tmp3 + z

if (tmp2 <= tmp4) goto L
  a = 0
L: b = 2
```

---

**3AC representation**

- each instruction represented using a structure called a “quad”
  - space for the operator
  - space for each operand
  - pointer to auxiliary info (label, successor quad, etc.)
- chain of quads sent to an architecture-specific machine-code-generation phase

---

**Code Generation**

**For base**

- skip building a separate IR
- generate code by traversing the AST
  - add codeGen methods to AST nodes
  - directly emit corresponding code into file

**Two high-level goals**

- generate correct code
- generate *efficient* code
Code Generation (cont.)

Simplified strategy
Make sure we don't have to worry about running out of registers
• for each operation, put all arguments on the stack
  • make use of the stack for computation
  • only use two registers for computation

Different AST nodes have different responsibilities
Many nodes simply "direct traffic"
• ProgramNode.codeGen
• List-node types
• DeclNode
  • TupleDeclNode
  • FctnDeclNode
  • VarDeclNode

Code Generation for Global Variable Declarations

Source code:
integer name.
tuple MyTuple instance.

In AST: VarDeclNode

Generate:
.data
.align 2 # align on word boundaries
_name: .space N # N is the size of variable

Size of variable
• for scalars, well-defined: integer, boolean are 4 bytes
• for tuples: 4*size of tuples
Code Generation for Function Declarations

**Need to generate**

- preamble
- prologue
- body
- epilogue

---

### MIPS Crash Course

#### Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sp</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>return address</td>
</tr>
<tr>
<td>$v0</td>
<td>used for system calls and to return int values from function calls, including the syscall that reads an int</td>
</tr>
<tr>
<td>$f0</td>
<td>used to return double values from function calls, including the syscall that reads a double</td>
</tr>
<tr>
<td>$a0</td>
<td>used for output of int and string values</td>
</tr>
<tr>
<td>$f12</td>
<td>used for output of double values</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>temporaries for ints</td>
</tr>
<tr>
<td>$f0 - $f30</td>
<td>registers for doubles (used in pairs; i.e., use $f0 for the pair $f0, $f1)</td>
</tr>
</tbody>
</table>
MIPS Crash Course (cont.)

Program structure

Data
- label: `.data`
- variable names & size; heap storage

Code
- label: `.text`
- program instructions
- starting location: `main`

Data

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vl:</td>
<td>.word</td>
<td>10</td>
</tr>
<tr>
<td>a1:</td>
<td>.byte</td>
<td>'a', 'b'</td>
</tr>
<tr>
<td>a2:</td>
<td>.space</td>
<td>40</td>
</tr>
</tbody>
</table>

40 here is allocated space – no value is initialized

Memory instructions

\texttt{lw}  
\texttt{register\_destination, RAM\_source}
- copy word (4 bytes) at source RAM location to destination register.

\texttt{lb}  
\texttt{register\_destination, RAM\_source}
- copy byte at source RAM location to low-order byte of destination register

\texttt{li}  
\texttt{register\_destination, value}
- load immediate value into destination register

\texttt{sw}  
\texttt{register\_source, RAM\_dest}
- store word in source register into RAM destination

\texttt{sb}  
\texttt{register\_source, RAM\_dest}
- store byte in source register into RAM destination
MIPS Crash Course (cont.)

Arithmetic instructions

\begin{align*}
\text{add} & \quad t0,t1,t2 \\
\text{sub} & \quad t2,t3,t4 \\
\text{addi} & \quad t2,t3, 5 \\
\text{addu} & \quad t1,t6,t7 \\
\text{subu} & \quad t1,t6,t7 \\
\text{mult} & \quad t3,t4 \\
\text{div} & \quad t5,t6 \\
\text{mfhi} & \quad t0 \\
\text{mflo} & \quad t1
\end{align*}

Control instructions

\begin{align*}
\text{b} & \quad \text{target} \\
\text{beq} & \quad t0,t1,\text{target} \\
\text{blt} & \quad t0,t1,\text{target} \\
\text{ble} & \quad t0,t1,\text{target} \\
\text{bgt} & \quad t0,t1,\text{target} \\
\text{bge} & \quad t0,t1,\text{target} \\
\text{bne} & \quad t0,t1,\text{target} \\
\text{j} & \quad \text{target} \\
\text{jr} & \quad t3 \\
\text{jal} & \quad \text{sub\_label} \quad \# \quad \text{"jump and link"}
\end{align*}

Check out: MIPS tutorial

https://minnie.tuhs.org/CompArch/Resources/mips_quick_tutorial.html