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

```
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 refering to

Two ways to set up dynamic access

- deep access – somewhat similar to access links
- shallow access – somewhat similar to displays

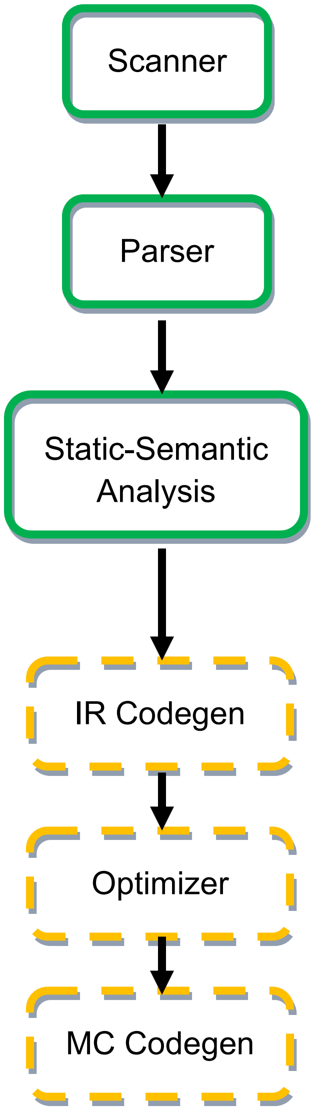
Deep access

- 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



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 op z
- x = op y
- x = y

Indirection
- x = y[z]
- y[z] = x
- x = &y
- x = *y
- *y = x

Call/Return
- param x,k
- retval x
- call p
- enter p
- leave p
- return
- retrieve x

Type Conversion
- x = AtoB y

Jumps
- if (x op y) goto *L*

Labeling
- label L

Basic Math
- times, plus, etc.

Example

source code

```
if  x + y * z > x * y + z [
    a = 0.
]
b = 2.
```

3AC code

```
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 auxilary info (label, succesor 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

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

MIPS Crash Course (cont.)

Program structure

Data

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

Code

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

Data

```
             name:   type      value(s)
e.g.,
             v1:     .word     10
             a1:     .byte     'a' , 'b'
             a2:     .space    40
```

40 here is allocated space – no value is initialized

Memory instructions

`lw register_destination, RAM_source`

- copy word (4 bytes) at source RAM location to destination register.

`lb register_destination, RAM_source`

- copy byte at source RAM location to low-order byte of destination register

`li register_destination, value`

- load immediate value into destination register

`sw register_source, RAM_dest`

- store word in source register into RAM destination

`sb register_source, RAM_dest`

- store byte in source register into RAM destination

MIPS Crash Course (cont.)

Arithmetic instructions

```
add     $t0,$t1,$t2
sub     $t2,$t3,$t4
addi    $t2,$t3, 5
addu    $t1,$t6,$t7
subu    $t1,$t6,$t7

mult    $t3,$t4

div     $t5,$t6

mfhi    $t0
mflo    $t1
```

Control instructions

```
b       target
beq     $t0,$t1,target
blt     $t0,$t1,target
ble     $t0,$t1,target
bgt     $t0,$t1,target
bge     $t0,$t1,target
bne     $t0,$t1,target

j       target
jr      $t3

jal     sub_label       #  "jump and link"
```

Check out: MIPS tutorial

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