CS536 Final Exam Review

Reid Chen

<2021-12-09 Thu>

Contents

1 Topics on the Final Exam

1.1 Semantic Analysis

- Name analysis / symbol table
- static scope vs. dynamic scope
- variable overloading
- type checking
  - What are types? $\mathbb{Z}$ = set of all integers

  - type system
    * primitive types
    * compound types
    * typing rules
  - type errors
  - static typing
    * checked by compiler
    * check at compiler type
  - dynamic typing
    * check at runtime
  - strong vs. weak typing
    * the degree to which a language catch a type error at compile/run time

1.2 Runtime Environment

- How the memory is laid out
  - the stack
    * store temporary things
    * activation records (AR)
  - the heap
    * store data dynamically allocated data
  - caller-callee relationship
    * parameter passing
      · pass by value
      · make a copy of the value
      · pass by reference
      · pass the address
      · pass by value-result

1.3 Code Generation

- Intermediate Representation (IR, pseudo assembly code)
- Compiler Backend

- MIPS
- Code generation in MIPS
  - AST $\rightarrow$ MIPS
  - `if`, `while`, function calls, returns, etc
- Control-flow graph (CFG)
  - Flowchart representation of the code
  - Each node is a basic block
    - If you execute the beginning of a basic block, then you execute the end of the basic block.
    - Thus, can perform optimization in a basic block

1.4 Optimization

- Difficulty
- Peephole optimization
  - Canonical optimization
    1. `mult by 2` $\rightarrow$ `shift by 1`
- Loop invariant code motion (LICM)
- Constant propagation
- Dead-code elimination

1.5 Data-flow Analysis

- Live variable analysis
- Reachable definition
- Static Single Assignment (SSA)

2 Code Generation

2.1 Spim

Special Registers in Spim

Register	Purpose
`$sp`	stack pointer
`$fp`	frame pointer
`$ra`	return address
`$v0`	used for system calls, return `int` from function calls
`$f0`	used for return `double` from function calls
`$a0`	used for output of `int` and `string`
`$f12`	used for output of `double`
`$t0 - $t7`	registers for `int`
`$f0 - $f30`	registers for `double`

2.2 Code Generation for Global Variable Declarations

Let `N` denote the size of the global variable in bytes. For each global variable `_v`, we generate the following code snippet:

```
    .data     # in static data area
    .align 2  # align on a word boundary
_v: .space N  # label N bytes area with name _v
```

2.3 Code Generation for Functions

Need to generate the following 4 parts in order:

1. preamble
2. entry
3. body
4. exit

2.3.1 Generating Function Preamble

For the `main` function,

```
    .text
    .globl main
main:
```

For all other functions,

```
.text
_<functionName>:
```

where `<functionName>` is a placeholder for the function called `functinoName`. The `.text` indicates the assembler that the instructions below should be stored in the text area.

2.3.2 Generating Function Entry

We do not need to worry about the actual parameters because the caller of this function has already pushed them on the stack. We need to do the following 4 things in order:

1. push the return address

   ```
   sw   $ra, 0($sp)
   subu $sp, $sp, 4
   ```

2. push the control link

   ```
   sw   $fp, 0($sp)
   subu $sp, $sp, 4
   ```

3. set the `$fp`

   ```
   addu $fp, $sp, 8
   ```

4. push space for local variables

   ```
   subu $sp, $sp, <size of locals in bytes>
   ```

   where `size of locals in bytes` can be calculated during semantic analysis.

2.3.3 Generating Function Body

No need to generate code for declarations, but need to generate code for each statement.

1. Write Statement

(a) Call the `codeGen` of the expressions in the write statement so that the value to be written will be placed on the top of the stack.

```
myExp.codeGen();
```

(b) Generate code that pop the value one the top of the stack into `$a0`

```
genPop(a0, 4);
```

(c) Set `$v0` to 1

```
generate("li", v0, 1);
```

(d) Make a system call

```
generate("syscall");
```

2. If-Then Statement

   (a) Evaluate the condition

   (b) Pop the top-of-stack value into register `$t0`

   (c) Jump to `FalseLabel` if `$t0` is `FALSE`

   (d) Code for the statement list

   (e) `FalseLabel:`

3. If-Then-Else Statement

   (a) Evaluate the condition

   (b) Pop the top-of-stack value into register `$t0`

   (c) Jump to `FalseLabel` if `$t0` is `FALSE`

   (d) Code for the then-statement list

   (e) Jump to `Exit`

   (f) `FalseLabel:`

(g) `Exit:`

4. While Statement

   (a) `Start:`

   (b) Evaluate the condition

   (c) Pop the top-of-stack value into register `$t0`

   (d) Jump to `FalseLabel` if `$0` is `FALSE`

   (e) Code for the statement list

   (f) `Start`

   (g) `FalseLabel:`

5. Return Statement

6. Read Statement

7. Identifier

2.3.4 Generating Function Exit

Want to pop off this function's AR. Then jump to the address that stored in the return address field of this function's AR. Popping off this function's AR means to restore the `$sp` and `$fp` to its caller' values. However, instead of simply setting `$sp` to `$fp`, we want to store `$fp` to a temporary register. Then restore `$fp` using the value stored in the control link field. Lastly, we restore `$sp` using the value stored in that temporary register. We restore `$sp` because a system interrupt may happen and use the stack. If we restore `$sp` at the beginning, the system interrupt may overwrite data we need.

```
lw   $ra, 0($fp)
move $t0, $fp
lw   $fp, -4($fp)
move $sp, $t0
jr   $ra
```