CSC617M Programming Language Theory

Marc Oliver Sison, John Regalado, Abien Fred Agarap
marc_sison@dlsu.edu.ph, john_go_regalado@dlsu.edu.ph, abien_agarap@dlsu.edu.ph

Overview

This document elaborates the four different approaches in implementing our interpreter system.

Interpreter

- All variables/parameters/return addresses are implemented on a stack.
- Uses Java objects to simplify storage of variables on stack; to eliminate the need for specifying different sizes, including array types.
- Two available "registers", not used for variable storage:
  ip - current line (0 index) of the three address code file (.tac)
  ebp - frame pointer (0 index). Also stored on the stack for recursion.
- All variables/parameter locations are offsets from ebp to find their location on the stack.
- Variable/parameter information of all functions/scope are in the .symb file

What happens on a function call:

1. Parameters are pushed on the stack.
2. *ip+1* is pushed on the stack which is return the address.
3. Ip is set to the location of the function
4. *ebp* is pushed on the stack.
5. Current stack pointer is pointed to *ebp*.
6. Space is allocated onto the stack to store local variables.

How function exit works including returns:

1. Local variables are deallocated on the stack.
2. *ebp* is popped from the stack.
3. Pop the return address from the stack and set ip to return address.
4. If return value exists, value is pushed to the stack after popping the return address.

What happens after the function call:

1. Return value is popped from the stack.
2. Parameters are popped from the stack.

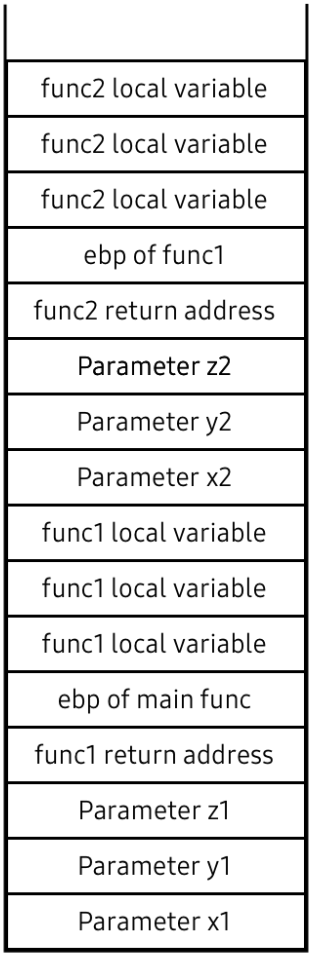
Sample Stack

main

call func1(x1,y1,z1) - *function has 3 local variables*

call int func2(x2,y2,z2) - *function has 3 local variables*

Stack before exiting func2:



Built-In Functions

`readString()`

Finds and returns the next complete token from this scanner.

Return type: `String`

`readInt()`

Scans the next token of the input as an Integer.

Return type: `Integer`

`readFloat()`

Scans the next token of the input as a float.

Return type: `Float`

`readChar()`

Scans the next token of the input as a character.

Return type: `Character`

`readBool()`

Scans the next token of the input as a boolean.
Return type: `Boolean`

`printString(string s)`
Prints a string.
Return type: `Void`

`printInt(int n)`
Prints an integer.
Return type: `Void`

`printFloat(Float f)`
Prints a float.
Return type: `Void`

`printChar(char c)`
Prints a character.
Return type: `Void`

`printBool(bool b)`
Prints boolean.
Return type: `Void`

Syntax

This section defines the syntax for our defined programming language.

- No global variables
- Supported unary operators `- !`
- Supported binary operators `+ - * / <= < > >= == !=`
- No explicit cast operation
- Implicit casts between `float` to `int` and `int` to `float`. Casts to left operand, i.e. `int + float`, `float` is cast to `int` before operation. Casts on assign
- `!` only applies to boolean
- unary - only applies to `int` and `float`
- Binary operators only apply to same types except `float` and `int` which will be implicitly casted.
- Curly braces for conditional statements (`if...else`) and loop statements (`while` / `do` / `for`) are required.
- `i++`, `++i`, and `i+=1` are not supported.
- Modulo (`%`) operation is not supported.
- Function prototypes must be declared. However, unlike C++, it is not necessary for the variable names declared in the function prototypes to match the variable names declared in function declaration.

`void`

No value returned.

`bool`

Returns boolean values, i.e. `true` and `false`

`float`

primitive `float` is `int` that contains a dot (`.`)

auto casts to `int` when applicable

`int`

auto casts to `float` when applicable

`char`

'c'

`string`

"string"

`arrays`

Passed by reference while everything else are passed by value

No array return type by functions

No multidimensional array

No array reassignment

Array size must be declared

```
int x[10];
int x[] = {1, 2, 3, 4};
```

Semantic error might be thrown in case of `int x[1+2]` but not in `int x[size1+size2]`

Supports variable array size, `int x[size]`

`const`

`const int`, `const int[]` declaration of immutable types.

`if(expression){ statements } else { statements }`

`expression` must be a conditional expression

`while(expression)`

`expression` must be a boolean expression

`do{}while(expression);`

`expression` must be a boolean expression

`for (assign; expression; assign)`

`expression` must be a boolean expression

assign are assignment statements i.e. i=0

no declarations at assign

single statements only

`Nested declaration`

inner declarations hide outer declarations

outer declarations are accessible from inner scopes

Optimization

We define the optimization methods implemented in our interpreter system.

- **Unreachable code optimization.** We create a control flow graph of basic blocks from the code, remove unreachable blocks from `main`. A basic block is a block of code where the entire block always executes sequentially in its entirety.
- **Local common subexpression optimization** (in a basic block). We create directed acyclic graphs from a basic block. When an expression can have multiple targets (multiple variables are equivalent), we prioritize non-temporary variables.
    - Rules for arrays when dumping the DAG back to code
        - Position of array reads to same array cannot cross array writes to same array
        - Position of array writes to same array cannot cross array writes to same array
        - Position of array reads to same array may pass array reads to same array
- **Local dead code optimization.** A side effect of local common subexpression optimization. We assume that only non-temporary variables are live on block exit and do not emit temporary variables if a non-temporary variable has already been emitted for that node.

Semantic Checker

Rules of syntax specify how language elements are sequenced to form valid statements. Thus, syntactic checking verifies that keywords, object names, operators, delimiters, and so on are placed correctly in your statements.

The following are the semantic errors our interpreter can handle.

- All semantic checks supports nested statements
- Undeclared variables
- Uninitialized variables
    - Only partially supported as the language could not identify initializations inside loops or control
- Using an expression as a statement. Statements are limited to function calls, control structures, and assignments.
- Variable used like a function
- Non-array variable used like an array
- Array variable used like a non array
- Non-integer index of array
- Array size declaration and initialization list size mismatch
    - Only partially supported as the language supports dynamic array size allocation
- Negative array initialization
    - Only partially supported as the language supports dynamic array size allocation
- Array bounds check
    - Only partially supported as the language supports dynamic array size allocation
- Type mismatch

- Function parameter mismatch
- Invalid casts
- Invalid types for binary operators
- Invalid types for unary operators
- Constant reassignment
- Function name reassignment
- Declaration of void type variable
- Double declaration on variables in the same scope
- Global variables not allowed
- Declaring a variable with the same name as a function
- Not declaring array size on declaration
- Expression statements not allowed
- Mismatch return type
- Missing return statement
- Undeclared function prototype
- Function already defined
- Function prototype and definition mismatch
- Function prototype exist but missing definition

Three Address Code

`var1 = var2`

simple assign

`var1 = var2 binary_op var3`

`binary_op - + * / < <= > >= == != || &&`

`var1 = unary_op var2`

`unary_op ! -`

`var1 = FLOAT_TO_INT var2`

generated because of autocast float to int

`var1 = INT_TO_FLOAT var2`

generated because of autocast int to float

`v1 SETSIZE v2`

- set size of array `v1` to `v2`, allocates space in the array object

`var1 = var2[var3]`

- array read

`var1[var2] = var3`

- array write

`Label:`

-GOTO locations

`GOTO Label`

change ip to location of label

`IF_FALSE var1 GOTO Label`

if var 1 is false change ip to location of label

`IF_TRUE var1 GOTO Label`

if var 1 is true change ip to location of label

`PUSHPARAM var`

-push var to stack

`POPPARAM num`

-pop num entries from stack

`v1 = POPPARAM`

-pop top of stack to v1

`CALL function_name`

- push ip+1 to the stack. change ip to function_name location

`BEGIN_FUNC`

- save previous frame pointer to stack. set current frame pointer to top of stack
- allocate space on stack for local variables

`END_FUNC`

-same as RETURN

`ENTER_SCOPE scope_name`

-enter scope_name in symbol table

`EXIT_SCOPE`

-return to previous scope

`RETURN`

deallocate local variables in stack.
load previous frame pointer from stack
pop return ip from stack
change ip to popped ip

`RETURN var1`

deallocate local variables in stack.
load previous frame pointer from stack
pop ip from stack
push var1 to stack
change ip to popped ip

Symbol table (`.symb` file) tree structure of tables

global table
| - function names and locations in code
| - label names locations in code
function
| - local variable names, types, locations on stack
| - scopes
| - local variables in this scope, types, locations on stack
| - scopes can nest

sourceMapper
- mapping of lines from TAC to original source
- no entries when code is optimized

Graphical User Interface

Supports code highlighting and code templates

Type keyword then hit `Ctrl+Shift+Space`

Keywords supported:

```
do
for
while
if
else
pl - printString("\n");
```

File menu -> open

Load a file

File menu -> save

Save contents of main window to a file

Run -> Run

Run the code without code optimization

Run -> Run optimized

Run the code with code optimization

Debug -> Run

Run the code without optimization. Break at breakpoints.

Popup local variables on break point

Right click -> Set Break Point

Add a break point for Debug -> Run

Right click -> Remove Break Point

Remove a break point for Debug -> Run