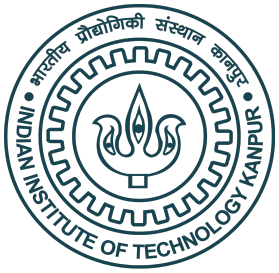

CS335 Compiler

MileStone 3

Swarnendu Biswas

April 4, 2023

Jaya Gupta (200471)

Harshit Bansal (200428)

Mohit Gupta (200597)

---

1 Requirements

- g++ $\geq$ 11, gcc $\geq$ 11 : The system should also have g++ installed.
- flex: The environment should have flex installed.
  - If its Linux then it can be installed from the command

```
sudo apt install flex
```

- Bison : The environment should have bison installed.
  - If its Linux then it can be installed from the command

```
sudo apt install bison
```

2 Execution Instructions

Compilation and Execution Instructions

```
make clean
make
./build/milestone/javair --input <input-file-name>
```

To parse all the testcases in tests folder, execute run.sh script.

Command Line Options

```
Usage: javair [-h] --input VAR --output VAR [--verbose]

Optional arguments:
  -h, --help    shows help message and exits
  -v, --version prints version information and exits
  --input       java file to parse [required]
  --output      output dot file [required]
  --verbose     increase output verbosity for parser
```

3 Three - AC Code

- Arrays and Class Instance Creation Objects are stored in the heap while its pointer stored in the stack using just 4 bytes of stack-space.
- The filename convention for the 3-AC is:
  Classname.Methodname_argumenttypeconcat.3ac
- Here argumenttypeconcat is _ seperated concatenation of argument types of arguments of functions along with the dimension size of that argument(zero for variables).
  Eg.: argumenttypeconcat is "int0_float2" for arguments (int a, float [][] b)
- In 3-AC code Labels are denoted by

```
        LabelName :
```

- The 3AC code for operations is written in the form

```
        operator argument1 argument2 result
        // For only 1 argument it leaves that argument2 empty

    Eg. Code:
        a = 1 + 2;
        b = 5;

    Eg. 3-AC:
        = 1 2 t1
        = t1 a
        = 5 t2
        = t2 b
```

- Array initialization and use instructions are done as follows. We have even allowed variable or even expressions as array dimensions.

```
Eg. Code:
    int b = 1, c = 3;
    double [][]a = new double [b+c][6];
    a[2][3] = 5;

E. 3-AC:
    = 1  t0
    = t0  b
    = 3  t1
    = t1  c
    +int b c t2
    = 1  t3
    * t3 t2 t4
    = t4  t3
    * t3 6 t4
    = t4  t3
    * t3 8 t4
    = t4  t3 // t3 stores memory used by array = (b+c)*6*8
    pushparam t3
    add esp   4      // space for arguments
    add esp   4      // space for array reference returned
    call allocmem 1
    mov  [esp + 0x0] t5      // get array reference
    sub esp   4      // remove space for array reference
    sub esp   4      // remove space for arguments
    = t5  t6
    = t6  a
    = 1  t7
    * t7 6 t8
    = t8  t7
    * t7 8 t8
    = t8  t7
    * t7 2 t9
    + a t9 t10
    = 1  t11
    * t11 8 t12
    = t12  t11
    * t11 3 t13
    + t10 t13 t14
    cast_to_double 5  t15
    = t15  t16
    = t16  *t14
```

- Also we have allowed object's instance arrays assignment. In that case the array address will be calculated by adding instance array offset to the object. As shown in below code.

Function Call, function declaration, Class instance creation and use instructions are done as follows. We have even allowed class instance creation before class declaration as is the case with actual Java 17 compiler.

Eg. Code:

```
class A{
    int[][] a = new int[5][5];
    A(int x){
        x = 5;
    }
}

class TestEmployee1 {
    public static void main() {
        // double [][]a = new double [5][6];
        // a[2][3] = 5;
        A obj = new A(5);
        int a = obj.a[2][3];
        obj.a[2][3] = 1;
    }
}
```

Eg. 3-AC:-

TestEmployee1.main.3ac :-

```
    beginfunc
    push ebp
    = esp ebp

    add  esp  8   // space for local variables

    pushparam 4
    add esp   4      // space for arguments
    add esp   4      // space for object reference returned
by allocmem
    call allocmem 1
    mov  [esp + 0x0] t0   // get object reference
    sub esp   4      // remove space for object reference
returned by allocmem
    sub esp  4       // remove space for arguments
    pushparam 5
    pushparam t0
    add   esp 8
    call A.A_int0 2
    sub   esp 8
    = t0  t1
    = t1  obj
```

```
      + obj 0 t2
      = 1  t3
      * t3 5 t4
      = t4  t3
      * t3 4 t4
      = t4  t3
      * t3 2 t5
      + *t2 t5 t6
      = 1  t7
      * t7 4 t8
      = t8  t7
      * t7 3 t9
      + t6 t9 t10
      = *t10  t11
      = t11  a
      + obj 0 t12
      = 1  t13
      * t13 5 t14
      = t14  t13
      * t13 4 t14
      = t14  t13
      * t13 2 t15
      + *t12 t15 t16
      = 1  t17
      * t17 4 t18
      = t18  t17
      * t17 3 t19
      + t16 t19 t20
      = 1  t21
      = t21  *t20

      ret:
       sub  esp  8 // manipulate stack pointer to the top of
stack removing local variables
       pop ebp
       return
       endfunc


 A.A_int0.3ac :-

      beginfunc
      push ebp
      = esp ebp

      add  esp  4   // space for local variables

      popparam x
      = 5  t4
```

```
        = t4   x

        ret:
         sub   esp   4 // manipulate stack pointer to the top of
    stack removing local variables
         pop ebp
         return
         endfunc
```

4 Runtime Support For Procedural Calls

4.1 Activation Record Fields:

Our activation record consists of the following fields:

- space for actual parameters
- space for return value
- space for old stack pointers to pop an activation. Our implementation as per x86 convention has two stack pointers, **esp** (Stack Pointer) and **ebp** (Base Pointer).
- space for locals

We have dealt with temporaries in 3AC, hence we have no knowledge of registers of the target architecture. Hence, no space is allocated in stack till now to save caller-saved registers.

4.2 Activation Record Visualisation

Let us consider the following program.

```
public class Demo {
    static int add(int a, int b){
        int x = a+b;
        return 5;

        int dead_var = 2;
    }

    static void main(){

        Demo obj = new Demo();
        int a = 20;
        float b = 40;
        char c = 'a';
```

```
        boolean bool = true;

        obj.add(1, 12);
    }
}
```

The Activation Record of the above code will look as follows

4.3 3AC instructions for Stack Manipulation and Activation Records

- **Pushparam** $\rightarrow$ **pushparam** $< param\_name >$
  - Just pushes the function parameters (¡param_name¿) onto the stack, does not manipulate stack pointer.
  - The Stack Pointer is explicitly manipulated using **add/sub** instructions as described below.

- In Milestone 4, this 3ac instruction will be replaced by **mov** instruction with explicit reference to memory location which will be calculated using **bp** and **offset**. The value will be stored by giving memory address location.

- **Popparam** $\rightarrow$ **popparam** $< param\_name >$
  - It is used to take values of arguments put in stack by the caller into variable (¡param_name¿) .
  - The stack pointer is not manipulated by this instruction.
  - In Milestone 4, this 3ac instruction will be replaced by **mov** instruction with explicit reference to memory location which will be calculated using **bp** and **offset**. The value will be read by giving memory address location.

- **Push** $\rightarrow$ **push** $< param\_name >$
  - Pushes the value(¡param_name¿) into stack and increment the stack pointer by 8.

- **Pop** $\rightarrow$ **pop** $< param\_name >$
  - Pops the value from the stack and stores it in reg/var(¡param_name¿), and decrement the stack pointer by 8

- **Add** $\rightarrow$ **add** $< reg >< add\_amt >$
  - Increments the register(stack pointer) by **add_amt**.

- **Sub** $\rightarrow$ **sub** $< reg >< add\_amt >$
  - Decrements the register(stack pointer) by **add_amt**.

- **Mov** $\rightarrow$ **mov src dest**
  - Moves the value from src register or memory location to dest register or memory location
  - In Milestone4, for every move, we will replace it by **movl** or **movq** instructions -¿ depending on size of moving value (long or int or float ..)

- **call** $\rightarrow$ **call** $< func\_name >< total\_args\_count >$
  - Calls the function labelled by func_name with total_args_count number of parameters. In our 3ac as well as in x86, it pushes the return address into the stack.

- **return**
  - Returns from the function to the return address stored in callee activation record by the caller.

5 Assumptions

- Array Initialization in curly braces {2,4,5} is not supported. Array can only be initialized by new keyword.
- System.out.println and Strings are not supported. String Type is not supported.
- Scope is kept as a number. It is incremented when a new scope(function, for, if-else, while, do while) is seen and decremented as we exit from these scopes. This is done for ease of usage and does not affect the performance or accuracy of parser and semantic analyser in any case.
- Each variable scope is assigned a number. So in cases like below:

```
if(true){
    int x;
} else {
    int x;
}

// Here in symbol table dump, there will be two similar entries
   as both x have same name, modifiers, type and same scope(as
   scope is a number). Both if and else have same scope number
   as per the policy in point above.
```

- Our 3AC looks slightly different from the 3AC example of piazza due to some notational convention adopted by us as described in the section Three-AC Code above.