CS 335 Semester 2022–2023-II: Project Milestone

Dishay Mehta (200341) Shrey Mehta (200580) Shubhan R (200971)

April 4, 2023

README file:
The environment should have flex and bison installed.
If its Linux then it can be installed from the command "sudo apt install flex".
If its Linux then it can be installed from the command "sudo apt install bison".
The system should also have g++ installed.

Compilation and Execution

Go to the required directory by command "cd src"
Enter "make" on the command line to build the src folder
We have created a wrapper code to make the milestone3
To execute,
Enter the command "cd /path/to/root"
Then the command "./milestone3 -i <inputfile>"
Note: You are required to change each and every file in rwx mode, i.e. do chmod 777 to each and every file in case of access denial.

Output

We have the output generated in our terminal itself.
This output will be the 3ac code for the input java testcode.

Compile Instructions

The following command line options have been implemented by us

- -i : The input flag where the input file must be fed.
- -v : the verbose option which provides the debugging info in case of any errors in the code.
- -h : The help option which lists the execution instructions.

Implementation Details

Three Address Code

The final 3ac code is dumped and the symbol table is printed on the terminal. We have used Quadruple representation of the 3AC code in the implementation. A struct is made to store this and a vector of this struct is maintained and the 3AC code corresponding to each expression is pushed into this vector, which are dumped in CSV files for each functions at the end. We have added the 3AC code corresponding to each and every function following the following format:

- Expressions:

```
a = b * c + b * d;
```

```
_t1 = b * c;
_t2 = b * d;
_t3 = _t1 + _t2;
a = _t3;
```

- Array Access:

```
void Binky(int[] arr)
{
   arr[1] = arr[0] * 2;
}
```

```
_Binky:
    BeginFunc 44;
    _t0 = 1;
    _t1 = 4;
    _t2 = _t1 * _t0;
    _t3 = arr + _t2;
    _t4 = 0;
    _t5 = 4;
    _t6 = _t5 * _t4;
    _t7 = arr + _t6;
    _t8 = *(_t7);
    _t9 = 2;
    _t10 = _t8 * _t9;
    *(_t3) = _t10;
    EndFunc;
```

- If-Else Statements

```
if (a < b + c)
     a = a - c;
c = b * c;
```

```
    _t1 = b + c;
    _t2 = a < _t1;
    IfZ _t2 Goto _L0;
    _t3 = a - c;
    a = _t3;
_L0: _t4 = b * c;
    c = _t4;
```

- Loops (While loop shown, similarly for others)

```
void main()
{
   int a;
   a = 0;

   while (a < 10) {
      Print(a % 2 == 0);
      a = a + 1;
   }
}
```

```
main:
   BeginFunc 40;
   _t0 = 0;
   a = _t0;
_L0:
   _t1 = 10;
   _t2 = a < _t1;
   IfZ _t2 Goto _L1;
   _t3 = 2;
   _t4 = a % _t3;
   _t5 = 0;
   _t6 = _t4 == _t5;
   PushParam _t6;
   LCall _PrintBool;
   PopParams 4;
   _t7 = 1;
   _t8 = a + _t7;
   a = _t8;
   Goto _L0;
_L1:
   EndFunc;
```

- Method Invocation

```
int foo(int a, int b)
{
   return a + b;
}

void main()
{
   int c;
   int d;

   foo(c, d);
}
```

```
_foo:
   BeginFunc 4;
   _t0 = a + b;
   Return _t0;
   EndFunc;
main:
    BeginFunc 12;
    PushParam d;
    PushParam c;
    _t1 = LCall _foo;
    PopParams 8;
    EndFunc;
```

- ClassInstanceCreation

```
class Animal {
  int height;
  void InitAnimal(int h) {
    this.height = h;
  }
}

class Cow extends Animal {
  void InitCow(int h) {
    InitAnimal(h);
  }
}

void Binky(Cow betsy) {
  betsy.InitCow(5);
}
```

```
_Animal.InitAnimal:
   BeginFunc 0;
      *(this + 4) = h;
      EndFunc;
VTable Animal =
       _Animal.InitAnimal,
;
_Cow.InitCow:
   BeginFunc 8;
      _t0 = *(this);
      _t1 = *(_t0);
   PushParam h;
   PushParam this;
      ACall _t1;
   PopParams 8;
      EndFunc;
 VTable Cow =
       _Animal.InitAnimal,
       _Cow.InitCow,
;
_Binky:
   BeginFunc 12;
      _t2 = 5;
      _t3 = *(betsy);
      _t4 = *(_t3 + 4);
   PushParam _t2;
   PushParam betsy;
      ACall _t4;
   PopParams 8;
      EndFunc;
```

Corresponding to these rules, the 3AC code has been made. Beginning of class and function have been indicated by BeginClass and BeginFunc respectively in the .3ac file. We have also implemented the cast_to_int etc.

Stack

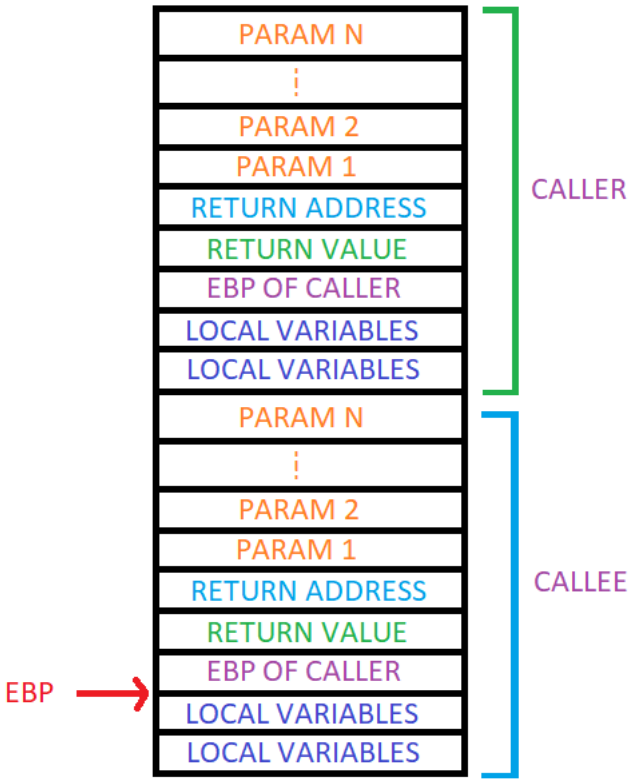

Figure1: Downward growing stack

The activation record of a function has the above structure in our compiler. We have written the code such that *"param"* instruction is used to push the parameters of a function before calling it. This does not change stack pointer. *"popparam"* instruction is used to read the top value of stack and also does not change the stack pointer. Therefore, the above 2 instructions require separate stack manipulation. *"push"* and *"pop"* instructions are used only to store and restore the value of EBP register from the stack and also store the return value in EAX register and also push it to the stack. They change the stackpointer on their own. The following steps occur in order whenever a function is called:

- **Precall code in Caller:**
  - Push all parameters of the callee in reverse order of how they are passed using *param* instruction.
  - Decrement the stack pointer by the size of parameters pushed so that it moves from pointing to the bottom of caller activation record to pointing to the bottom of pushed parameters.
  - Calling the callee adds the return address to the stack and decrements stack pointer accordingly.
- **Prologue code in Callee:**
  - Saves the old value of EBP register in the stack and decrements it.

- Before any local variable declaration, the stack pointer is decremented to make space for the variable.

- **Epilogue code in Callee:**
  - Increment the value of bytes that were decremented to make space for local variables.
  - The value on top of the stack is EBP of caller now. We pop and restore this.
  - Push the return value to the top of the stack. So basically, the EBP of caller that was popped in the previous step and the return value occupy the same address in the stack.
  - Then control returns to caller.

- **Postcall code in Caller:**
  - The stackpointer points to the return value stored. This is popped and stored in the caller. The stackpointer now points to the bottom of the parameters list of the callee.
  - The stackpointer is now incremented to point to the bottom of the frame of the caller.

- **Other Functions Used:**
  - **add a b:** Adds value b to a
  - **sub a b:** Subtracts value b from a
  - **call Funcname:** Calls function Funcname which will be replaced by jump instruction in x86

To showcase our implementation of stack, let us take an example of a very simple function code:

```
int f(int a,int b)
{
   int x;
   return a;
}
```

b
a
RA
BP
x
↓

Table 1: Stack for this function

Assumptions

- We have assumed that only the types present in raw JAVA language are allowed. The types like ”String” are not supported by us.

- We throw type-checking errors which are the static compilation checks done while parsing the grammar.

- The Three AC code has been made according to the rules above and also according to the examples given on piazza (This 3AC encoding is a reference from Handout written by Maggie Johnson and revised by Julie Zelenski).
- We have added support to access modifiers like public, private, static and final only just as mentioned on Piazza.
- We have not given errors for the variables which are assigned without initialisation, as that is a compiler developer dependent property, as has been mentioned in the JAVA Documentation.
- Kindly ignore any errors due to System.out.println() in the 3AC Code.
- All the basic features as mentioned on Piazza have been incorporated in the submission.
- Even if the file may be erroneous, the symbol table dumps and the 3AC code will be generated.
- Kindly ensure that the class names don't start with string "Class" and Methods don't start with "Method" for dumping purposes.
- All the functions must be declared before used, i.e. you cannot call a function defined later before calling it as we wanted to generate all the specified functionalities in one parse through the input string. If required, we may perform 2 parses, in which in the first parse, we make the symbol table and in the second parse, we can do the typechecking and generate the Three AC Code.

References

Handbook written by Maggie Johnson and revised by Julie Zelenski
Stanford Compiler Design Lecture 13