CS 335 Semester 2022–2023-II: Project Milestone

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

March 23, 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 myASTgenerator
To execute,
Enter the command "cd /path/to/root"
Then the command "./myASTgenerator -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

The symbol tables have been dumped using the convention :

- Classes : Class<classnname>.csv
- Methods : Method<methodname>.csv

The Three Address Code corresponding to each method is dumped using the convention :

- <classname>.<methodname>.3ac

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

Symbol Table

- We have maintained two classes named Sym_Entry and Sym_Table, the first class storing the fields of the symbols and the Sym_Table storing a map which is the symbol table, the parent pointer, which points to the parent of that scope of the symbol table, and the level number of that symbol table.
- The symbol table which has scope named "Global" is the global scope of the file being parsed, and has level = 0 and parent pointing to NULL.
- The symbol table has fields: Lexeme, Token, Type, Offset, Scope Name and Line Number. The other fields which are present in the class were used for implementing specific functionalities.
- The corresponding semantic actions were added in each production, such that the scope is changed and a new symbol table is formed, each time we enter a new scope and the previous table is stored in a vector of symbol tables which can later be accessed by their scope names for type checking.

Type Checking

We have run many experiments on real Java compilers to understand type checking rules to make sure our type checking rules agree with standards. We have also read many online websites regarding how classes and objects and static etc. work. We have done type checking for all expressions and operators, methods, arrays, classes. Even static, public, private and final type checking as specified on the forum are implemented. We have added appropriate error messages with the line numbers also. **System.out.println may give error in some code lines as this as not been implemented yet and the instructor has told us to ignore it on Piazza. Please consider the error due to this accordingly.**
We have also implemented type-casting as asked in the description of the milestone 2.

Three Address Code

The final 3ac code is printed in the terminal. We have also dumped the files. In case any of the dump files are missing please consider the terminal one as final. 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.

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 have throwing errors for the basic 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, which is a different representation as compared to the example 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.
- The arrays which are allocated using new are allocated in the heap memory, so they do not have any offset, since their dimensions may be variable and hence are defined in dynamic memory and not static memory.

References

Handbook written by Maggie Johnson and revised by Julie Zelenski