Flat-B Compiler - BCC

Susobhan Ghosh

Flat-B Language Description

Flat-B is a simple language, similar to C.

Syntax & Semantics

All the variables have to be declared in the declblock{....} before being used in the codeblock{...}. Multiple variables can be declared in the statement and each declaration statement ends with a semicolon.

```
declblock
{
	...
}
codeblock
{
	...
}
```

Data Types: Only Integers and Array of Integers are supported

```
int data, array[100];
int sum;
```

Statement Types:

1. **Assignment expression:** Variable assignment is fundamental to any programming language. Syntax is given below. Math expression could be binary mathematical operations on variables or constants alike.

   ```
   variable = math_expression;
   a[i] = 2;
   b[a[i+j]] = c[d[e[3] + i]] + a[i] + 5;
   ```

2. **For loop** : For loop is supported. As per the syntax below, the upper limit given is inclusive. By the current definition, one cannot have a reverse loop, as the condition check for loop is always `variable < upper limit`

   ```
   for variable = math_expression, upper limit, increment = 1 {
   	.....
   }

   for i = 1, 100 {
   	.....
   }

   for i = 1, 100, 2 {
   	.....
   ```

```
}
```

3. **if-else statement**: If-else statements are supported. As per the syntax below, the expression has to be a boolean binary or unary compare expression, but it will never take values or variables, or even boolean keyword "true" or "false". More on conditional expressions below

```
if conditional_expression {
        ....
}
....

if a <= b {
        ....
}
....

if conditional_expression {
        ....
} else {
        ....
}
```

4. **While loop**: While loop expression is supported. The syntax is given below

```
while conditional_expression {
        ....
}

while a != b {
        ....
}
```

5. **Input/Output Statements**: Print and Read statements are supported. Syntax given below

```
print "blah...blah", val;
println "new line at the end";
read sum;
read data[i];
```

6. **Conditional and Unconditional Goto Statements**: Conditional goto occur with an if statement - and jump occurs only if the condition evaluates to true, while unconditional goto makes the control jump directly to the label. Labels should be defined only once throughout the program. Labels can be put in front of any of statements mentioned above [1-5]

```
label: statement;
...
goto label;
goto label if conditional_expression;
```

Expression Types:

1. **Mathematical Expression:** A mathematical expression supports four operands - Add, Subtract, Multiply and Divide. It also supports brackets - (and). A math expression could be a proper math expression with operators in between other math expressions, variables and / or constants.

2. **Conditional Expression:** A conditional expression is an expression to test/condition two mathematical expressions. It supports six binary operations for test - geq (>=), leq (<=), >, < , EQTO (==) and NEQ (!=) - and one unary operation - NOT (!).

Context Free Grammar

```
program:            declblock { declaration } codeblock { statements }
                    | declblock { } codeblock { statements }
                    | declblock { declaration } codeblock { }
                    | declblock { } codeblock { }

declaration:        decl_line
                    | declaration decl_line

decl_line:          int midentifiers ;

midentifiers:       identifierdecl
                    | midentifiers , identifierdecl

statements:         statements IDENTIFIER : statement_line
                    | IDENTIFIER : statement_line
                    | statements statement_line
                    | statement_line

identifier:         IDENTIFIER '[' math_expression ']'
                    | IDENTIFIER

identifierdecl:     IDENTIFIER '[' NUMBER ']'
                    | IDENTIFIER

math_expression:    math_expression + math_expression
                    | math_expression - math_expression
                    | math_expression * math_expression
                    | math_expression / math_expression
                    | ( math_expression )
                    | NUMBER
                    | - NUMBER
                    | - identifier
                    | identifier

assignment:         identifier = math_expression

cond_statement:     math_expression <= math_expression
                    | math_expression >= math_expression
                    | math_expression > math_expression
                    | math_expression < math_expression
                    | math_expression == math_expression
                    | math_expression != math_expression
                    | ! cond_statement

gotoblock:          goto IDENTIFIER if cond_statement
                    | goto IDENTIFIER

statement_line:     assignment ;
                    | forloop
                    | whileloop
```

```
                        | ifelse
                        | iostatement ;
                        | gotoblock ;

forloop:                for assignment , math_expression , math_expression { statements }
                        | for assignment , math_expression { statements }

whileloop:              while cond_statement { statements }

ifelse:                 if cond_statement { statements } else { statements }
                        | if cond_statement { statements }

iostatement:            print STRINGID , math_expression
                        | println STRINGID , math_expression
                        | print STRINGID
                        | println STRINGID
                        | print math_expression
                        | println math_expression
                        | read identifier
```

AST Design

Below is the list of classes used in the AST Design of the Compiler, along with inheritance and some detail about their utility and members:

1. **ASTNode** - This is the virtual base class for all the following AST Classes. It has a virtual visit() function conforming to the visitor design pattern.
2. **ASTProgram** - Inherits ASTNode. This is the AST Class for the entire program. It stores references to the Declaration Block and Code Block.
3. **ASTDeclBlock** - Inherits ASTNode. This is the AST Declaration Block Class. It holds references to a list of Declaration Statements.
4. **ASTDeclStatement -** Inherits ASTNode. This is the AST Declaration Statement Class. It holds references to a list of variable sets (set of all variables declared in one line) to be used in the program as declared in the declblock{}
5. **ASTVariableSet -** Inherits ASTNode. This stores a set of all variables in a particular line of declblock{}, and the type (integer).
6. **ASTVariable** - Inherits ASTNode. This stores the single variable's name/string, the type (integer), and the information whether this variable is an array or not. If it's an array, it also stores the length specified.
7. **ASTCodeBlock** - Inherits ASTNode. This stores the list of code statements in a scope/codeblock.
8. **ASTCodeStatement** - Inherits ASTNode. This is a virtual class for all types of code statements as indicated in the CFG. Has a label member which is inherited by all code statement child classes.
9. **ASTIOBlock** - Inherits from ASTCodeStatement. Stores the type of instruction I/O (read or write), and the string to output (if specified), and the reference to associated expression/variable.
10. **ASTGotoBlock** - Inherits from ASTCodeStatement. Stores the goto label to jump to. Also stores reference to condition if it's an conditional goto.
11. **ASTIfElse** - Inherits from ASTCodeStatement. Stores reference to the condition for the if part, and references to **if** code block and else code block (if it exists)
12. **ASTWhileLoop -** Inherits from ASTCodeStatement. Stores the reference to the while loop condition, and the codeblock which executes inside the while loop.

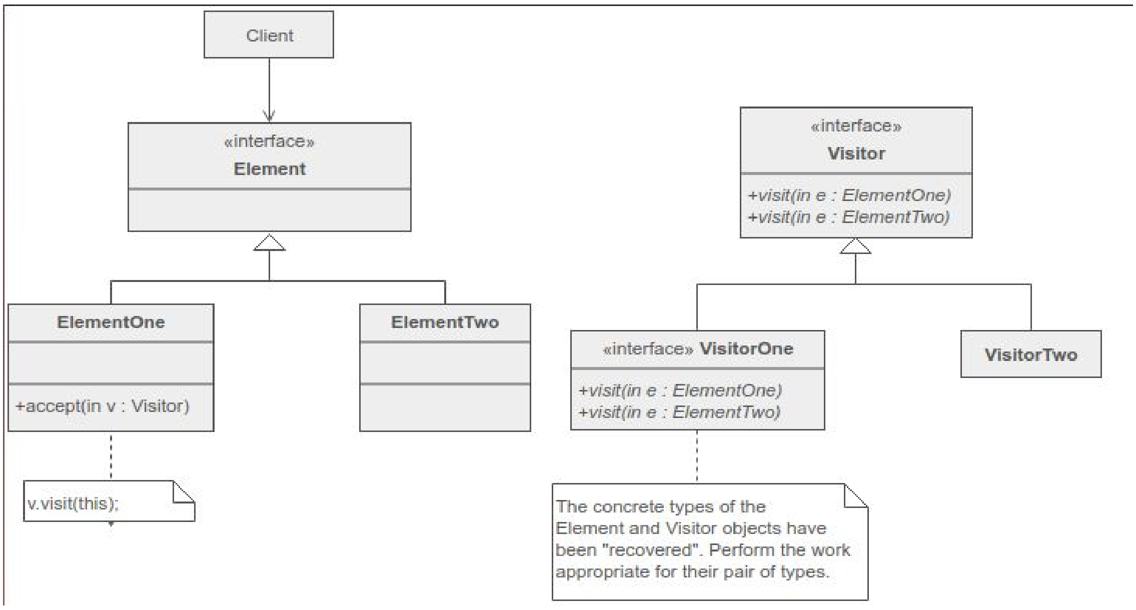
13. **ASTForLoop -** Inherits from ASTCodeStatement. Stores the reference to assignment statement, the loop condition, the increment statement, and the codeblock which executes inside the for loop.
14. **ASTAssignment -** Inherits from ASTCodeStatement. Stores the reference to the variable location where the assignment or store instruction will happen, and the reference to the math expression which will be stored at that location.
15. **ASTCondExpr** - Inherits ASTNode. Stores information about conditional expression, which means references to the left and right mathematical expression, and the conditional operator to be applied.
16. **ASTMathExpr -** Inherits ASTNode. Stores information about mathematical operation, which means references to the left and right mathematical expression, and the mathematical operator to be applied.
17. **ASTTargetVar** - Inherits from ASTMathExpr. Stores the location of the variable where the assignment statement will happen or where this variable's value is required. Stores a boolean value to indicate if the object is intended for store instruction or load instruction.
18. **ASTInteger** - Inherits from ASTMathExpr. Stores a constant integer value.

Visitor Design Pattern

When many distinct and unrelated operations need to be performed on Node objects in a heterogeneous aggregate structure, visitor design is an appropriate design to be used.

It is good design practice and also logical to avoid polluting the Node classes with distinct and unrelated operations. One doesn't have to query the type of each node and cast the pointer to the correct type before performing these desired operations.

Visitor design looks somewhat like this (image taken from sourcemaking.com) -



So, Node classes here are the Element classes. Each Element class inherits from the base virtual Element Class, and has an accept() method. The Visitor Interface is also shown, which is again a virtual base class with a visit() method. The accept() method in each Element class and derivedclass takes a Visitor class as argument, and calls the Visitor's visit() function. This visit() method takes as parameter all types of Element classes (as seen in Visitor Interface, refer diagram). So, if a new Visitor is made (say VisitorTwo), one can easily perform a different set of functions using this class, over the same Element classes, without touching the Element class definitions. This essentially achieves the

separation of class unrelated operations, and enables easy plug and play mechanism of changing operations.
This compiler (BCC) also uses the Visitor Design pattern, with a base virtual class called the **Visitor**. The Visitor interface which traverses the entire AST inherits from this Visitor base class, is the **ASTVisitor**.
The accept() function in each ASTNode is somewhat like -

```
void accept(Visitor *)
```

And the ASTVisitor has the following set of visit() functions -

1. `void visit(ASTIOBlock *);`
2. `void visit(ASTGotoBlock *);`
3. `void visit(ASTIfElse *);`
4. `void visit(ASTCondExpr *);`
5. `void visit(ASTForLoop *);`
6. `void visit(ASTWhileLoop *);`
7. `void visit(ASTMathExpr *);`
8. `void visit(ASTInteger *);`
9. `void visit(ASTTargetVar *);`
10. `void visit(ASTAssignment *);`
11. `void visit(ASTCodeBlock *);`
12. `void visit(ASTVariable *);`
13. `void visit(ASTVariableSet *);`
14. `void visit(ASTDeclStatement *);`
15. `void visit(ASTDeclBlock *);`
16. `void visit(ASTProgram *);`

Each of these visit() functions traverse the nodes to make the AST, and generate the AST in a XML file called the AST_XML.xml.

Interpreter Design

The Interpreter is made using simple C++ code execution, and is achieved by another interface inheriting from **Visitor**, called the **ASTInterpreter**. It also has the same set of visit() functions which **ASTVisitor** has (as defined virtually in **Visitor**). It has a few additional visit_value() functions, which I had to add to return values from ASTNodes (since the visit() functions were all declared as void return types). The additional functions added were -

1. `bool visit_value(ASTCondExpr *);`
2. `int  visit_value(ASTMathExpr *);`
3. `int  visit_value(ASTTargetVar*);`
4. `int  visit_value(ASTInteger  *);`
5. `void visit_value(ASTTargetVar*, int);`

Corresponding accept_value() functions were also added to 4 classes - ASTCondExpr, ASTMathExpr, ASTInteger, and ASTTargetVar - to call these visit_value() functions.

LLVM CodeGen Design

The CodeGen design again uses the similar visitor design pattern, like the two above. The class associated is the **CodeGenVisitor**. But, due to the fact that all the visit() functions in the **Visitor** class were void return types (barring few used in ASTInterpreter), it made more sense to not make CodeGen inherit from the **Visitor** base class. Visitor Design Pattern is still used, but **CodeGenVisitor** doesn't inherit from Visitor. New codegen() functions for each ASTNode has been declared (which is symbolical to accept() in visitor design pattern) with the prototype - `Value* codegen(CodeGenVisitor*);`

Value is a LLVM Class. A Value pointer used for returning almost any type of LLVM instruction. For the **CodeGenVisitor**, the following visit methods are declared -

```
1.  Value* visit(ASTIOBlock            *);
2.  Value* visit(ASTGotoBlock          *);
3.  Value* visit(ASTIfElse             *);
4.  Value* visit(ASTCondExpr           *);
5.  Value* visit(ASTForLoop            *);
6.  Value* visit(ASTWhileLoop          *);
7.  Value* visit(ASTMathExpr           *);
8.  Value* visit(ASTInteger            *);
9.  Value* visit(ASTTargetVar          *);
10. Value* visit(ASTAssignment         *);
11. Value* visit(ASTCodeBlock          *);
12. Value* visit(ASTVariable           *);
13. Value* visit(ASTVariableSet        *);
14. Value* visit(ASTDeclStatement      *);
15. Value* visit(ASTDeclBlock          *);
16. Value* visit(ASTProgram            *);
```

The rest of the design is same as the visitor design pattern - ASTNode's codegen() returns the value of CodeGenVisitor's visit(ASTNode).

Performance Comparison

Bubble Sort - I ran a bubblesort program, to sort an array of 10000 in ascending order, with numbers from 1 to 10000. The array contained the numbers in the descending order - 10000, 9999, ..., 1. I generated .ll and .bc files for the program, and below are the performance results using lli, llc and the interpreter -

Performance counter stats for 'lli-3.9 bubblesort.b.bc':

```
      235.652842      task-clock (msec)         #    0.998 CPUs utilized
               0      context-switches          #    0.000 K/sec
               0      cpu-migrations            #    0.000 K/sec
           1,547      page-faults               #    0.007 M/sec
    65,06,26,225      cycles                    #    2.761 GHz
    38,96,53,953      stalled-cycles-frontend   #   59.89% frontend cycles idle
 <not supported>      stalled-cycles-backend
    87,66,37,850      instructions              #    1.35  insns per cycle
                                                #    0.44  stalled cycles per insn
    15,52,30,461      branches                  #  658.725 M/sec
        1,91,449      branch-misses             #    0.12% of all branches

     0.236214907 seconds time elapsed
```

Performance counter stats for 'llc-3.9 bubblesort.b.bc':

```
       14.756106      task-clock (msec)         #    0.973 CPUs utilized
               0      context-switches          #    0.000 K/sec
               0      cpu-migrations            #    0.000 K/sec
           1,556      page-faults               #    0.105 M/sec
     3,28,02,085      cycles                    #    2.223 GHz
     2,22,77,003      stalled-cycles-frontend   #   67.91% frontend cycles idle
 <not supported>      stalled-cycles-backend
```

```
      2,69,70,080      instructions              #    0.82  insns per cycle
                                                 #    0.83  stalled cycles per insn
        53,23,250      branches                  #  360.749 M/sec
         1,86,768      branch-misses             #    3.51% of all branches

      0.015167076 seconds time elapsed
```

Performance counter stats for 'lli-3.9 bubblesort.b.ll':

```
       224.536252      task-clock (msec)         #    0.998 CPUs utilized
                0      context-switches          #    0.000 K/sec
                0      cpu-migrations            #    0.000 K/sec
            1,547      page-faults               #    0.007 M/sec
      65,21,19,265      cycles                   #    2.904 GHz
      38,99,32,019      stalled-cycles-frontend  #   59.79% frontend cycles idle
   <not supported>      stalled-cycles-backend
      87,68,32,559      instructions             #    1.34  insns per cycle
                                                 #    0.44  stalled cycles per insn
      15,52,90,241      branches                 #  691.604 M/sec
         1,95,480      branch-misses             #    0.13% of all branches

      0.225034245 seconds time elapsed
```

Performance counter stats for '../src/bcc bubblesort.b':

```
     144840.360178      task-clock (msec)        #    1.000 CPUs utilized
               157      context-switches         #    0.001 K/sec
                33      cpu-migrations           #    0.000 K/sec
             1,216      page-faults              #    0.008 K/sec
  3,84,18,16,99,828      cycles                  #    2.652 GHz
  1,71,86,16,20,273      stalled-cycles-frontend #   44.73% frontend cycles idle
    <not supported>      stalled-cycles-backend
  5,74,42,54,87,242      instructions            #    1.50  insns per cycle
                                                 #    0.30  stalled cycles per insn
  1,08,59,00,60,113      branches                #  749.722 M/sec
    2,46,21,28,667      branch-misses            #    2.27% of all branches

     144.860876313 seconds time elapsed
```

We can see that, lli running on the .bc (bytecode) and the .ll file is produces almost the same results. The wall-clock times are almost same at ~0.23s, and number of instructions are also almost the same at 876 million. llc on the other hand executes very fast, at ~0.015s, and number of instructions is way less at 26 million. Lastly, our own implementation of interpreter performs very badly. It takes 144 seconds, and 108 billion instructions. Although this includes the lexer and scanner, the time taken, and the instructions are largely contributed by the Interpreter itself (lexer and scanner had finished instantly, but sorting wasn’t, as control was not back to the terminal).

Selection Sort - Similar to the above case, I ran an selection sort program this time to sort an array of 10000 in ascending order, with numbers from 1 to 10000. The array contained the numbers in the descending order - 10000, 9999, ..., 1. I generated .ll and .bc files for the program, and below are the performance results using lli, llc and the interpreter

Performance counter stats for 'lli-3.9 selectionsort.b.bc':

```
     159.346196      task-clock (msec)         #    0.997 CPUs utilized
              0      context-switches          #    0.000 K/sec
              0      cpu-migrations            #    0.000 K/sec
          1,549      page-faults               #    0.010 M/sec
    39,95,92,907      cycles                   #    2.508 GHz
    23,54,09,652      stalled-cycles-frontend  #   58.91% frontend cycles idle
  <not supported>     stalled-cycles-backend
    50,23,75,991      instructions             #    1.26  insns per cycle
                                               #    0.47  stalled cycles per insn
    13,03,77,716      branches                 #  818.204 M/sec
       2,55,283      branch-misses             #    0.20% of all branches

    0.159873110 seconds time elapsed
```

Performance counter stats for 'llc-3.9 selectionsort.b.bc':

```
      19.661405      task-clock (msec)         #    0.983 CPUs utilized
              0      context-switches          #    0.000 K/sec
              0      cpu-migrations            #    0.000 K/sec
          1,562      page-faults               #    0.079 M/sec
     2,75,21,695      cycles                   #    1.400 GHz
     1,61,50,897      stalled-cycles-frontend  #   58.68% frontend cycles idle
  <not supported>     stalled-cycles-backend
     2,75,33,900      instructions             #    1.00  insns per cycle
                                               #    0.59  stalled cycles per insn
       54,48,697      branches                 #  277.127 M/sec
       1,93,694      branch-misses             #    3.55% of all branches

    0.020011531 seconds time elapsed
```

Performance counter stats for 'lli-3.9 selectionsort.b.ll':

```
     162.862038      task-clock (msec)         #    0.997 CPUs utilized
              2      context-switches          #    0.012 K/sec
              1      cpu-migrations            #    0.006 K/sec
          1,552      page-faults               #    0.010 M/sec
    40,34,59,030      cycles                   #    2.477 GHz
    24,10,46,789      stalled-cycles-frontend  #   59.75% frontend cycles idle
  <not supported>     stalled-cycles-backend
    50,26,61,759      instructions             #    1.25  insns per cycle
                                               #    0.48  stalled cycles per insn
    13,04,52,972      branches                 #  801.003 M/sec
       2,65,148      branch-misses             #    0.20% of all branches

    0.163340053 seconds time elapsed
```

Performance counter stats for './../src/bcc selectionsort.b':

```
     56003.167635      task-clock (msec)         #    1.000 CPUs utilized
               57      context-switches          #    0.001 K/sec
                7      cpu-migrations            #    0.000 K/sec
            1,215      page-faults               #    0.022 K/sec
 1,61,57,70,57,608      cycles                    #    2.885 GHz
   70,55,19,48,511      stalled-cycles-frontend   #   43.66% frontend cycles idle
   <not supported>      stalled-cycles-backend
 2,73,88,73,14,984      instructions              #    1.70  insns per cycle
                                                 #    0.26  stalled cycles per insn
   52,26,41,13,135      branches                  #  933.235 M/sec
     35,18,10,947      branch-misses             #    0.67% of all branches

     56.010141826 seconds time elapsed
```

Again, in this case, llc performs the best, clocking 0.02 seconds and 27 million instructions. lli comes second clocking 0.16 seconds and 502 million instructions. Last comes our interpreter, clocking 56 seconds and 273 billion instructions.

<u>Sieve Of Eratosthenes:</u> I ran a program of sieve of Eratosthenes, to find all primes below 1000000. I generated .ll and .bc files for the program, and below are the performance results using lli, llc and the interpreter-

Performance counter stats for 'lli-3.9 sieveoferatosthenes.b.bc':

```
        54.223872      task-clock (msec)         #    0.990 CPUs utilized
                0      context-switches          #    0.000 K/sec
                0      cpu-migrations            #    0.000 K/sec
            1,949      page-faults               #    0.036 M/sec
      9,83,05,070      cycles                    #    1.813 GHz
      8,08,35,103      stalled-cycles-frontend   #   82.23% frontend cycles idle
  <not supported>      stalled-cycles-backend
      4,59,38,180      instructions              #    0.47  insns per cycle
                                                 #    1.76  stalled cycles per insn
        85,43,489      branches                  #  157.560 M/sec
         1,74,845      branch-misses             #    2.05% of all branches

      0.054778999 seconds time elapsed
```

Performance counter stats for 'llc-3.9 sieveoferatosthenes.b.bc':

```
         9.614146      task-clock (msec)         #    0.971 CPUs utilized
                0      context-switches          #    0.000 K/sec
                0      cpu-migrations            #    0.000 K/sec
            1,552      page-faults               #    0.161 M/sec
      2,97,11,741      cycles                    #    3.090 GHz
      1,92,62,586      stalled-cycles-frontend   #   64.83% frontend cycles idle
  <not supported>      stalled-cycles-backend
      2,56,43,458      instructions              #    0.86  insns per cycle
                                                 #    0.75  stalled cycles per insn
        50,38,190      branches                  #  524.039 M/sec
         1,74,749      branch-misses             #    3.47% of all branches

      0.009896270 seconds time elapsed
```

Performance counter stats for 'lli-3.9 sieveoferatosthenes.b.ll':

```
      45.375659      task-clock (msec)         #    0.991 CPUs utilized
              0      context-switches          #    0.000 K/sec
              0      cpu-migrations            #    0.000 K/sec
          1,943      page-faults               #    0.043 M/sec
    8,83,99,176      cycles                    #    1.948 GHz
    7,08,56,002      stalled-cycles-frontend   #   80.15% frontend cycles idle
<not supported>      stalled-cycles-backend
    4,53,05,244      instructions              #    0.51  insns per cycle
                                               #    1.56  stalled cycles per insn
      84,46,791      branches                  #  186.152 M/sec
       1,68,386      branch-misses             #    1.99% of all branches

    0.045801534 seconds time elapsed
```

Performance counter stats for './../src/bcc sieveoferatosthenes.b':

```
    1797.816483      task-clock (msec)         #    1.000 CPUs utilized
              4      context-switches          #    0.002 K/sec
              1      cpu-migrations            #    0.001 K/sec
          2,180      page-faults               #    0.001 M/sec
 5,53,70,92,200      cycles                    #    3.080 GHz
 2,49,75,07,478      stalled-cycles-frontend   #   45.11% frontend cycles idle
<not supported>      stalled-cycles-backend
10,10,93,91,652      instructions              #    1.83  insns per cycle
                                               #    0.25  stalled cycles per insn
 1,89,38,87,470      branches                  # 1053.438 M/sec
       1,31,119      branch-misses             #    0.01% of all branches

    1.798401264 seconds time elapsed
```

Again, in this case, llc performs the best, clocking 0.009 seconds and 25 million instructions. lli comes second clocking ~0.05 seconds and 45 million instructions. Last comes our interpreter, clocking 1.8 seconds and 10 billion instructions.