Sudheer Achary
(20161076)

Compilers Project

Flat-B Language Description:

Flat-B language mainly consists of declaration block where all the used variables are declared, code block consists of program logic expressed in multiple arithmatic expressions, assignments, different kind of loops.

Flat-B Syntax:

1. **Data Types**:

Inegers and Array of Integers.

Format:
```
int <variable>
int <variable> [ <number> ];
```

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

2. **Declaration Block**:

All the variables have to be declared in the declblock{....} before being used

3. Code Block:

in the codeblock{...}. Multiple variables can be declared in the statement and each declaration statement ends with a semi-colon.

4. **Expressions**:

Addition, Subtraction, Multiplication, Division, Mod operations are supported on integers of any arithmatic expressions.

examples:
```
v = 1 + 2;
v = a - 1;
v = 2*b / a;
v = A[i] * B[j];
v = A[i] / A[i+1];
```

5. **For Statement**:

Types of for-loops supported

Format:

Type-1:

```
for i = <start_expression>, <end_expression> {
     .....
}
```

Type-2:

```
     for i = <start_expression>, <end_expression>,
<step_expression> {
          .....
     }
```

6. **If-Else Statement**:

Format:

```
if <condition> {
     ....
}

if <condition> {
     ...
}
else {
     ....
}
```

7. **While Statement**:

Format:

```
while <condition> {

}
```

8. **Print Statement**:

Format:

```
print <variable>
print <string>
print <string>, <variable>
println <string>
```

9. **Read Statement**:

Format:

```
read <variable>
```

Flat-B Semantics:

- All the variables must be declared before hand for the usage.
- Only integer data type is supported for all variables and arrays.
- In For loop intial value of loop_variable should always be less than or equal to final value.
- Increment on for loop can be any positive constant (or) one.

Flat-B CFG examples:

```
v = 1 + 2;

=> <Location> EQ <Location> + <Location> SC

=> <Location> EQ <Expression> SC

=> <Assignment Expression>
```

```
if a == b {
     print “a b are Equal”;
}
else {
     print “a b are Not Equal”;
}

=> IF <Location> EQUALS <Location> OP PRINT <string> CP
     ELSE OP PRINT <string> CP

=> IF <Condition> OP <Statement> CP
     ELSE OP <Statement> CP

=> IF <Condition> <If_Block> ELSE <Else_Block>

=> <IfElseStatement>
```

```
for i = 1, j {
     print “value is : ”, j
}

=> FOR <Location> EQ <Location> COMMA <Location> OP
     PRINT <string> COMMA <Location>
     CP

=> FOR <Location> EQ <Expression> COMMA <Expression> OP
     <Statement>
     CP
```

```
=> FOR <Location> EQ <Expression> COMMA <Expression>
     <For_Block>

=> <For_Statement>


while i < 100 {
     Print i;
}

=> WHILE <Location> L <Location> OP
          PRINT <Location>
     CP

=> WHILE <Condition> OP <Statement> CP

=> WHILE <Condition> <While_Block>

=> <While_Statement>
```

Flat-B Abstract Syntax Tree Design:

All nodes in abstract syntax are of type <ASTNode>.

Each of the classes get inherits from <ASTNode>.

```
<ASTNode>
|
|----<Variable>
|
|----<FieldDeclaration>
|
|----<Statement>
|
|----<Main>
|
|----<Location>
|
|----<Block>
```

Each of the classes get inherits from <Statement>.

```
<Statement>
|
|----<For_Statement>
|
|----<While_Statement>
|
|----<If_Else_Statement>
|
|----<Print_Statement>
|
```

```
|----<Read_Statement>
|
|----<Assignment_Statement>
|
|----<Expression_Statement>
```

Complete AST Class Hierarchy

```
<ASTNode>
|
|----<Variable>
|
|----<Field_Declaration>
|
|----<Statement>
|         |
|         |----<For_Statement>
|         |
|         |----<While_Statement>
|         |
|         |----<If_Else_Statement>
|         |
|         |----<Print_Statement>
|         |
|         |----<Read_Statement>
|         |
|         |----<Assignment_Statement>
|         |
|         |----<Expression_Statement>
|----<Main>
|
|----<Location>
|
|----<Block>
```

AST Tree in XML format:

```
<program>
     <declarations>
          <declaration>
               <vars>
                    <var />
                    <var />
               </vars>
          </declaration>
          <declaration>
               <vars>
                    <var />
                    <var />
               </vars>
          </declaration>
     </declarations>

     <statements>
          <statement>
```

```
        <Assignment>
            <LHS />
            <Expr>
                <Binary Expression />
            </Expr>
        </Assignment>
    </statement>
    <statement>
        <Labeled Statement>
            <Label />
            <Statements></Statements>
        </Assignment>
    </statement>
    <statement>
        <IfStatement>
            <Expr>
                <BooleanExpression />
            </Expr>
            <IfBlock>
                <Statements></Statements>
            </IfBlock>
        </IfStatement>
    </statement>
    <statement>
        <IfElseStatement>
            <Expr>
                <BooleanExpression />
            </Expr>
            <IfBlock>
                <Statements></Statements>
            </IfBlock>
            <ElseBlock>
                <Statements></Statements>
            </ElseBlock>
        </IfElseStatement>
    </statement>
    <statement>
        <ForStatement>
            <LHS />
            <CondExpr>
                <BooleanExpression />
            </CondExpr>
            <StartExpr>
                <BinaryExpression />
            </StartExpr>
            <EndExpr>
                <BinaryExpression />
            </EndExpr>
            <StepExpr>
                <BinaryExpression />
            </StepExpr>
            <ForBlock>
                <Statements></Statements>
            </ForBlock>
        </ForStatement>
```

```
        </statement>
        ........
        ........
        ........
    </statements>
</program>
```

Flat-B Interpreter Design:

Each of the node in Flat-B AST can be interpreted by function call

```
int <ClassName>::interpret()
```

Flat-B Interpreter parses the grammar, interprets each node and runs corresponding action in C++.

Interpreter has two symbol tables for variables, array

variable symbol table

```
map <string, int> var_table
```

array symbol table

```
map <string, vector <int> > array_table
```

FieldDeclaration::interpret() -- declares & stores variables in var_table or array_table.

Location::interpret() -- returns value from var_table or array_table.

Expression::interpret() -- returns value after evaluating expression.

Assignment::interpret() -- evaluates rhs expression in assignment and stores in lhs Location.

While::interpret() -- evaluates <While_Block> till <While_Condition> fails.

For::interpret() -- evaluates <For_Block> for given <loop_variable> from <start> to <end> with given <step>.

IfElse::interpret() -- evaluates <If_Condition> if true evaluates <If_Block> otherwise evaluates <Else_Block>

Flat-B LLVM Code Generation:

Each of the Node in AST has <ClassName>::codegen() method that generates IR for a given <ASTNode>.

Main::codegen() ---> generates complete IR for declaration block
& code block.

Location::codegen() ---> generates IR for a number, if location
is variable it loads and returns a
type Value *.

Assignment::codegen() ---> generates IR for assignment statement,
it generates IR for evaluating rhs
and loading its value then stores in rhs.

Varible::codegen() ---> generates IR for variables as global i32 .

FieldDeclaration::codegen() ---> generates IR for variable
initialization with zero globally.

Read::codegen() ---> reads variable by inserting scanf like
function in IR

Print::codegen() ---> prints variable by inserting printf like
function in IR

Performance Comparision:

generating binary representation of a decimal number:

./bcc.sh ../test-units/bits.b 0.03s user 0.06s system 2% cpu
4.055 total

generating primes less than a given N:

./bcc.sh ../test-units/primes.b 0.04s user 0.04s system 2% cpu
2.817 total

sorting a given array of N number:

./bcc.sh ../test-units/bubblesort.b 0.02s user 0.03s system 0%
cpu 30.363 total