

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 arithmetic expressions, assignments, different kind of loops.

Flat-B Syntax:

1. Data Types:

Integers 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 arithmetic 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>

```

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 & returns a type Value*.

Assignment::codegen() ---> generates IR for evaluating rhs and
storing in lhs.

Variable::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.

IfElse::codegen() ---> creates 3 basicblock if_block, else_block,
merge_block inserts a breakpoint after
which the merge_block should be generated.

For::codegen() ---> creates for_block, cont_block basicblocks sets breakpoint to join cont_block on failure of condition.

While::codegen() --> creates while_block, cont_block basicblocks sets breakpoint to join cont_block on failure of condition.

Performance Comparision:

generate primes less than 100:

	time(ms)	instructions
Interpreter	29.196759	2,10,67,531
lli	52.731759	7,69,73,830

generate first 20 fibonacci numbers:

	time(ms)	instructions
Interpreter	52.698034	7,39,76,647
lli	30.195836	2,92,50,187

generate binary format for a decimal number:

	time(ms)	instructions
Interpreter	25.324382	2,04,03,226
lli	30.195836	2,92,50,187