

Report

FlatB Compiler

Ayushi Goyal



Contents

1. Language Description
2. Syntax and Semantics
3. Design of AST
4. Visitor design pattern
5. Design of Interpreter

1. Language Description

FlatB language contains 2 blocks, namely declblock and codeblock. Declblock contains all the variable declaration. Any variable not declared in this block cannot be used in the program. CodeBlock contains all the program statements such as loops, conditions, goto, assignment, read, print, evaluate, etc.

To leave the declblock without declaring any variable, the declaration list contains only ‘;’ .

```
declblock
{
    declaration_list
}
codeblock
{
    statement_list
}
```

Data Types

FlatB supports two data types: integer and array of integers.

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

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 semi-colon.

Identifiers

Identifiers can contain only alphabets or digits. Identifiers can start only with an alphabet.

Valid identifiers: data, data1, da1ta

Invalid Identifiers: if, for, else, 1data, data_1

Arithmetic Operators

Addition	+	a+b
Subtraction	-	a-b
Multiplication	*	a*b
Division	/	a/b
Modulo	%	a%b
Left shift	<<	a<<b
Right Shift	>>	a>>b

Equality operators

Equal	==	a==b
Not Equal	!=	a!=b

Conditional Operators

And	&&	a&&b
Or		a b

Relational Operators

Less than	<	a<b
Greater than	>	a>b

Less than and equal to	<=	a<=b
Greater than and equal to	>=	a>=b

Expressions

Expressions are of the type

Identifier operator identifier

Identifier

(expression)

These expressions are evaluated and can be used to assign to another variable or can be used in if condition or while looping condition.

If.. Else

The `if` statement is used for conditional execution:


If..Else construct has 3 variants: if, if..else, if..else if

```

if expression {
    ....
}

if expression {
    ....
}
else if expression {
    ....
}

```



```
if expression {  
    ....  
}  
else {  
    ....  
}
```

While loop

The [while](#) statement is used for repeated execution as long as an expression is true.

```
while expression {  
    ....  
}
```

For loop

The [for](#) statement is used to iterate over the elements of a sequence

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

Here 1 is the start value and the loop ends when i is greater than 100. 'i' should be declared in the declaration block.

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

Instead of specifying exact start and end values, variables containing these values can also be used.

Read and Print

Read statement is to take inputs from the user. Read statement can take only one variable as an input at a time, that is, read statement does not support cascading.

```
read a;  
read arr[9];  
read arr[i];
```

Print statement is used to output data on the stdout.

There are 2 constructs: print and println

println prints the data followed by a newline, whereas print does not follow the output with a newline.

Print supports cascading.

```
print "blah...blah", val  
println "new line at the end"
```

Goto Statements

It performs a one-way transfer of control to another line of code

Conditional Goto:

```
goto label if expression
```

Unconditional Goto

```
goto label
```

Syntax and Semantics

```
declblock[
```

```
Int a;
```

}

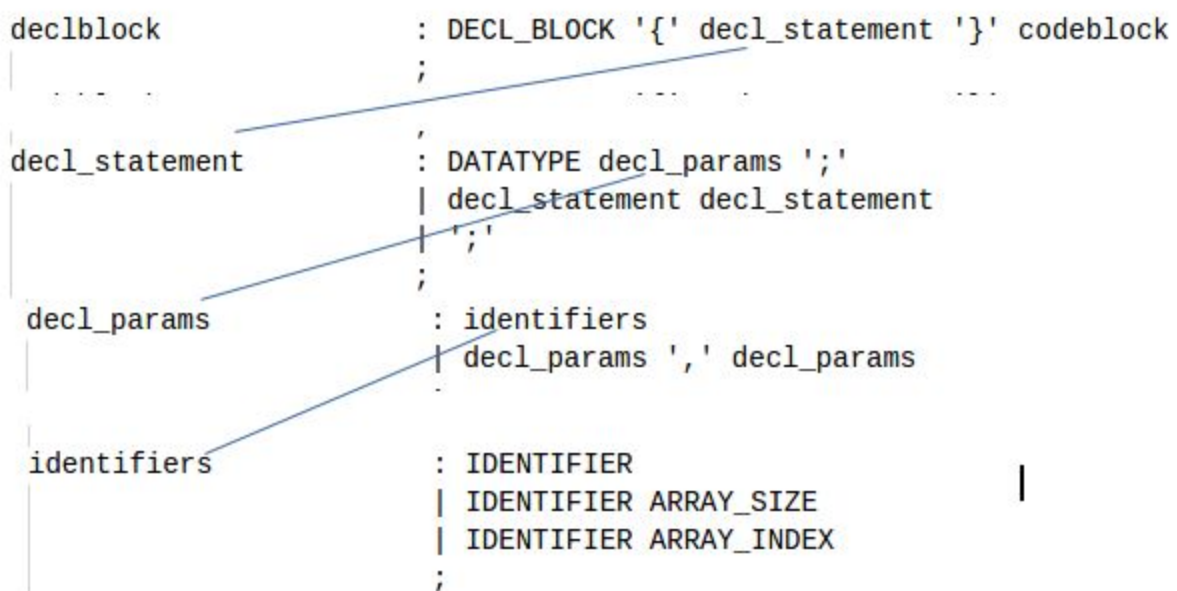
codeblock{

a=5;

}

For the given code snippet, derivation will be as follows:

The program starts at declblock



AST design

Each nonterminal element has a class defined

Each production rule for the nonterminal element is a child class to the nonterminal base class

Example:

<pre>else_statement : ELSE '{' code_statement '}' ELSE if_statement ;</pre>	<pre>{ASTElse* elsestmt = new ASTElse(\$3); \$\$ = new ASTElseStmt(); \$\$->ElseStmt = elsestmt;} {ASTElseIf* elseif = new ASTElseIf(\$2); \$\$ = new ASTElseStmt(); \$\$->ElseIf = elseif;}</pre>
---	--

For the given grammar, the AST classes are defined as:

```
//else statement
class ASTElseStmt:public ASTCodeStmt{
public:
    ASTElse* ElseStmt;
    ASTElseIf* ElseIf;
    ASTElseStmt()
    {
        this->ElseStmt=NULL;
        this->ElseIf=NULL;
    }
    void accept(Visitor* v)
    {
        v->visit(this);
    }
};
class ASTElse:public ASTElseStmt{
public:
    ASTCodeStmt *stmt;
    ASTElse(ASTCodeStmt* stmt)
    {
        this->stmt = stmt;
    }
    void accept(Visitor* v)
    {
        v->visit(this);
    }
};
class ASTElseIf:public ASTElseStmt{
public:
    ASTIfStmt *if_stmt;
    ASTElseIf(ASTIfStmt* if_stmt)
    {
        this->if_stmt = if_stmt;
    }
    void accept(Visitor* v)
    {
        v->visit(this);
    }
};
```

Here the base class ASTElseStmt corresponds to non-terminal element **else_statement**

This has 2 base classes each corresponding to it's production rule.

Visitor Design

Each AST class has an accept function defined which calls it's corresponding visit function

```
void accept(Visitor* v)
{
    v->visit(this);
}
```

The visit function then calls the accept function of the element inside it or the statement after it.

```
void Interpreter::visit(ASTMulticodeAssignment* astMulticodeAssignment)
{
    cout<<"multi code assignment"<<endl;
    astMulticodeAssignment->stmt->accept(this);
    astMulticodeAssignment->assign->accept(this);
}
```

This visit function calls the accept functions of the data member of the calling class

The visit functions are declared in class Visitor and defined in the class Interpreter

Interpreter

Interpreter class defined the visit function for every class.

This is where the compiler understands the code and executes it

```

class Interpreter : public Visitor
{
public :
    map<pair<string,int>,int > symbol_table;
public:
    void visit (class ASTNode* );
    void visit (class ASTDeclBlockNode* );
    void visit (class ASTParamsDeclStmt* );
    void visit (class ASTMultiDeclStmt* );
    void visit (class ASTDeclStmt* );
    void visit (class ASTDeclParams*);
    void visit (class ASTDeclMultiParams*);
    void visit (class ASTDeclIdParams*);
    void visit (class ASTCodeBlockNode*);
    void visit (class ASTCodeStmt*);
    void visit (class ASTCodePrint*);
    void visit (class ASTPrintStmt*);
    void visit(class ASTFinalPrintStmt*);
    void visit(class ASTMultiPrintStmt*);
    void visit(class ASTFinPrintStmt*);
    void visit(class ASTFinalPrintStmtId*);
    void visit(class ASTFinalPrintStmtText*);
    void visit(class ASTIdNode*);
    void visit(class ASTMultiCodePrint*);
    void visit(class ASTCodeAssignment*);
    void visit(class ASTAssignment*);
    void visit(class ASTMultiCodeAssignment*);
    void visit(class ASTCodeRead*);
    void visit(class ASTMultiCodeRead*);
    void visit(class ASTIfStmt*);
    void visit(class ASTMultiCodeIfStmt*);
    void visit(class ASTCodeIfStmt*);
    void visit(class ASTCodeIfElse*);
    void visit(class ASTMultiCodeIfElse*);
    void visit(class ASTElseStmt*);
    void visit(class ASTElse*);
    void visit(class ASTElseIf*);
    void visit(class ASTCodeFor*);
    void visit(class ASTMultiCodeFor*);
    void visit(class ASTForStmt*);
    void visit(class ASTCodeWhile*);
    void visit(class ASTMultiCodeWhile*);
    void visit(class ASTWhileStmt*);
    int evaluateExp(class ASTExp*);
    bool checkIdExist(string id);
};

```

This is an example of for_statement class and it's corresponding interpreter function

```
//for statement
class ASTForStmt:public ASTCodeStmt{
public:
    ASTForExp *exp;
    ASTCodeStmt *stmt;
    ASTForStmt(ASTForExp* exp, ASTCodeStmt* stmt)
    {
        this->exp = exp;
        this->stmt = stmt;
    }
    void accept(Visitor* v)
    {
        v->visit(this);
    }
};
```

```
void Interpreter::visit(ASTForStmt* astForStmt)
{
    cout<<"executing for loop"<<endl;
    string it = astForStmt->exp->id;
    int start = astForStmt->exp->num1;
    int last = astForStmt->exp->num2;
    int inc = astForStmt->exp->num3;
    cout<<"variable :: "<<it<<" "<<start<<" "<<last<<" "<<inc<<endl;
    if(symbol_table.find(make_pair(it, -1))!=symbol_table.end())
    {
        symbol_table[make_pair(it, -1)]=start;
        cout<<"iterator is declared and initialised to :: "<<symbol_table[make_pair(it, -1)]<<endl;
        for(symbol_table[make_pair(it, -1)] = start; symbol_table[make_pair(it, -1)]<=last; symbol_table[make_pair(it, -1)]+=inc)
        {
            astForStmt->stmt->accept(this);
        }
    }
    else{
        cout<<"variable not declared"<<endl;
        exit(0);
    }
}
```

■ ■ ■