PROJECT REPORT

Programming Lanugage Description:

1. Data Types
Inegers 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.

2. Expressions

3. for loop

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

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

4. if-else statement

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

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

....
}

5. while statment

while expression {

}

6. conditional and unconditional goto

got label

goto label if expression

7. print/read

print "blah...blah", val
read sum
read data[i]

Syntax and Semantics

1. Program starts from a non-terminal(NT) “program” which brances to NT “decl_blocks” and “code_blocks”. These two NT helps to parse the given style of language.

```
 program: DECLBLOCK '{' decl_blocks '}' CODEBLOCK '{' code_blocks
'}' {
  $$=new Prog($3,$7);
  start=$$;
};
```

For e.g, the simplest program in this language would look like

declblock{
}
codeblock{
}

where “decl_block” and “code_block” both terminates to *epsilon.*

2. NT “decl_blocks” is basically a list of NT “var” which further goes to terminal symbols(T) “IDENTIFIER” and “ARRAY”. Decl_list stores all the variables declared.

decl_block: INT variables {$$=new fieldDecl($2);};

variables: variable { $$=new Vars();$$->push_back($1);}
| variables ',' variable {
$$->push_back($3);
};

variable: IDENTIFIER {$$ = new Var(string("Identifier"),string($1));}
| ARRAY {$$=new Var(string("Array"),string($1));};

For e.g
declblock{
int x, arr[10];
int i, j;
}
codeblock{
}

3. Next, the parsing moves to NT “code_blocks” which, parses all the specified type of “code_block” possible which here are described by Nts “thingps”, “thingrs”, “expr”, “conds”, “forloop”, “call”. All the statements are ‘;’ terminated.

code_blocks: { $$=new fieldCodes(); }
| code_blocks code_block {$$->push_back($2);};

code_block:PRINT thingps ';' {$$=$2;}
| READ thingrs ';' {$$=$2;}
| expr ';' {$$=$1;}
| IF conds '{' code_blocks '}' {$$=new ifelsest($2,$4,NULL);}
| IF conds '{' code_blocks '}' ELSE '{' code_blocks '}' {$$=new ifelsest($2,$4,$8);}
| WHILE conds '{' code_blocks '}' {$$=new whilest($2,$4);}
| FOR forloop '{' code_blocks '}' {$$=new forst($2,$4);}

| LABEL code_blocks GOTO IDENTIFIER call ';' {$$=new gotost($2,$4,$5);}

Let's analyse them one by one with examples to understand the semantics better.

3(a) Print statements: Store the variables(could be identifiers, numbers, array, strings) in "printList" and then print them one by one.

```
    thingps:thingp {$$=new thingpsst();$$->push_back(string($1));}
  | thingps ',' thingp {$$->push_back(string($3));};
thingp:  IDENTIFIER {$$=$1;}
            | NUMBER      {$$=num2str($1);}
            | ARRAY       {$$=$1;}
            | STRING      {$$=$1;};
```

For e.g

```
    declblock{
    int x;
}
    codeblock{
    print x;
}
```

Similarly, READ works in the same manner.

```
     declblock{
     int x;
}
    codeblock{
    read x;
}
```

3(b): Expressions: On the right hand side, you have binary expressions with operators "+", "-", "*" and "/" which after evaluation goes to LHS, which could either be a identifier or an array address.

```
expr: IDENTIFIER '=' exprnew {$$=new expr($1,$3);}
            | ARRAY '=' exprnew {$$=new expr($1,$3);} ;

exprnew: arithmetic  {$$=new exprnewst($1); }
            | IDENTIFIER {$$=new exprnewst($1);}
```

| NUMBER {$$=new exprnewst($1);}
| ARRAY {$$=new exprnewst($1);};

arithmetic: exprnew ADD exprnew {$$=new arithmeticst($1,string($2), $3);}
| exprnew SUB exprnew {$$=new arithmeticst($1,string($2), $3);}
| exprnew DIV exprnew {$$=new arithmeticst($1,string($2), $3);}
| exprnew MUL exprnew {$$=new arithmeticst($1,string($2), $3);};

For e.g

```
declblock{
int x, y, arr[200];
}
codeblock{
x=2;
y=3;
arr[x]=y;
}
```

3(c) If else: Boolean expressions come into picture with NT “conds”.

conds: {$$=new condsst();} | conds andor cond {$$>push_back($3,$2);};
cond: exprnew compare exprnew {$$=new condst($1,$2,$3);};

A normal if-else block would look like

```
declblock{
int x, i, j, k;
}
codeblock{
    x=1;
    k=3;
    if(x<k)
    x=2;
```

```
}       else{
        x5;
}
}
```

3(d) While: Uses “conds” and “code_blocks”. Nothing new. Has structure like

```
        decl_block{
        int x;
}
codblock{
        x=2;
        while x<5{
        x=x+1;
}
```

3(e) For-Loop: Has a different structure than “WHILE”. NT “forloop” parses the loop using the grammar

forloop: IDENTIFIER '=' NUMBER ',' NUMBER inc {$$=new forloopinit($1,$3,$5,$6);};

where NT “inc” stores the increment value per iteration. If it’s NULL, then it’s taken to be 1.

For e.g

```
        declblock{
                int i, j, x;
        }
        codeblock{
                for i=0,10,3{
                x=x+i;
                }
}
```

3(f)Goto: The grammar is designed in a manner that the “LABEL” is only found upawards i.e the “code_blocks” for GOTO call is alread defined. NT “call” is used to specify whether the GOTO is conditional or unconditional.

LABEL code_blocks GOTO IDENTIFIER call ';' {$$=new gotost($2,$4,$5);}

call: {$$=NULL;}
| IF conds {$$=new callst($2);}

For e.g

```
declblock{
int x, i, j;
}
codeblock{

L1: x=x+1;
goto L1 if x< 10
}
```

Design of AST and Interpreter

AST is designed with root pointer store in “start”. It’s a Prog* type pointer which provides root to start all the traversals and codegenerations.
From here, it calls the traversals into declblocks and codeblocks. “decl_block” traversals intializes the variables whereas the “code_block” traversal traverses each and every code one by one. The “code_block” which is part of “codeb_locks” could be your specified blocks of code like for-loop, if-else etc.

When parsing is going on, corresponding action takes place which is used to pass the parsed language which is then used in the constructor. Each and every block has a class and respective constructor associated to them. With the parsing of each block, class construction comes into play and create an object for each class-type for later use in traversal and codegen.

Visitor Design Pattern

Implemented traversals and codegen using normal traversal. Didn’t use visitor design pattern.

Design of Codegen

Objects stored of each class to give a well-defined AST. Each class has a Value* codegen() which is then called from the object of that type. Each such codegen function inserts IRBuilder instructions as well as BasicBlocks to generate IR.

Performance Analysis

Wrote three programs->bubblesort, cumulative sum of array and finding factorial.

Bubblesort
lli
real 0m0.019s
user 0m0.012s
sys 0m0.004s

llc
real 0m0.192s
user 0m0.016s
sys 0m0.004s

Factorial
lli
real 0m0.010s
user 0m0.000s
sys 0m0.008s

llc
real 0m0.010s
user 0m0.008s
sys 0m0.000s

Cumulative Sum
lli
real 0m0.020s
user 0m0.008s

```
sys    0m0.008s

llc
real   0m0.011s
user   0m0.008s
sys    0m0.000s
```