COMS W4115

Advanced Arithmetic Language

May 10, 2012

Name: Jimin Choi
ID: jc3783

Introduction

To understand compiler/interpreter concepts learned from COMS W4115, I decided to implement an advanced arithmetic language. Unlike a basic 4-functions calculator, this language supports advanced mathematical operations such as roots, powers, average and summations, etc. This language is not a simple calculator. It supports basic programming language functionalities to allow users to implement algorithms. The functionalities are if-else statement blocks, for loops, while loops, functions and variables.

This language can be considered as a lightweight Matlab. The syntax for this language is not similar to C or Java. However, it has similar treatments to for/while loops, if-else statements, and functions as C. Due to the nature of the arithmetic language, it only supports integer variables and integer arrays. The target language for this compiler is Java. Ocaml was used to implement all compiler and language features.

Language Tutorial

A function in a programming language defines a set of routines to be executed. All statements including variable declarations and arithmetic operations in this language have to be defined inside functions.

Sample 1.

```
func void execute:{
        int i;
  for [i = 0 ; i < 5 ; i = i + 1] {
    print[i];
  }
}
```

This example defines a function. Any function declaration should start with a keyword “func” followed by a return type. In this case, “void” return type is used to indicate that no return value is needed. The function name is “execute”. The first line declares variable i, and a

for-loop is followed by the variable declaration. The for-loop iterates five times to print i incrementing i by one for each loop. All programs should have "func void execute" function as a main function. If your function needs to be executed, it should be called inside "execute".

Sample 2.

```
func void assignarray:array a, int b
{
  a(b) = 4;
}

func void execute:{
  array t(3);
  [assignarray:t, 2];
  print[t(2)];
}
```

This example defines a new function "assignarray" with two parameters a and b. "a(b) = 4" assigns 4 to the array a's element at index b. In the "execute" function, a new array t with a size 3 is declared on the first line. A function call is enclosed in square brackets. "[assignarray:t, 2]" is calling a function called assignarray with parameters t and 2. After the function call the second element is printed.

Sample 3.

```
func void execute:{
  int i;
  i = 1;
  if [i == 0] print[32]; else print[4];
  print[92];
}
```

This example demonstrates how to utilize conditional statements. if statement inside the function is checking the predicate "i == 0". If it is true, it prints 32 otherwise it prints 4.

How to compile your program

Requirement: Mac OS X, Linux, or Unix environment, jdk 1.6 or higher (lower versions may work but not tested)

1. Unzip amath.zip, and run make under amath folder.
2. Save your program file as (filename).am
3. When the build is complete, run this command.

   ./amath < (your file name with directory path) > AAL.java

4. Run java compiler by running

   javac AAL.java

5. Run “java AAL”

Language Reference Manual

A. Lexical Conventions

A.1. Identifiers

An identifier is consist of English alphabet letters, digits, and underscores. It should start with letters, and numbers can follow letters. Upper and lower case letters are considered as different.

A.2. Keywords

The following identifiers are reserved as keywords, and should not be used for other purposes.

- *func*
- *if*
- *else*
- *for*
- *while*
- *return*
- *int*
- *void*
- *array*
- *sqrt*
- *sum*
- *avg*
- *abs*
- *log*
- *print*

A.3. Constants

As a mathematical language, integer constant is supported. An integer is always considered as a decimal number.

A.4. Comments

The comment starts with /* and terminates with */. The comment cannot occur within character or integer literals.

B. Meaning of Identifiers

In this language, identifiers indicate functions or variables.

B.1 Functions

A function declaration starts with a “func” keyword. The return type of the function follows “func” keyword. The return type can be int, array, or void. The return type follows a function name. A colon is used to specify function arguments to the function. Each argument should specify a type and a name. If there are multiple function arguments to the function, a comma separates each parameter. Curly braces enclose the actual content of a function as a block.

func <type> <function-name>: <argument 1 type> <argument 1 name>, <argument 2 type> <argument 2 name> ... {
    <block of code>...
}

The scope of function arguments is within the function. When the function returns, the argument name is no longer valid. The scope of any variable defined inside the function has the same fate as function arguments.

B. 2 Variables

A variable is declared with a type followed by a name. Unlike function types, void type is not allowed for variable declarations. For integer variable, an assignment operator (=) is used to assign a constant to a variable.

<variable type> <variable name>;
<variable name> = <constant>;

An array variable can be defined slightly differently. Instead of a mere variable name, it should also specify the size of the array.

Array <variable name>(size);

C. Data Types

C.1 Integer

An integer (int) is a primitive type in this language. Most of calculation is based on integer constants. All integer values are decimal numbers.

C.2 Array

An array is used to store integers in consecutive memory spaces, and it is supported as a variable. Arrays in this language store integers only. An element in an array can be accessed using postfix parenthesis enclosing the index number.

C.3 Void

Void type (void) is used to return nothing in a function. Void variable is not allowed.

D. Expressions

The order of precedence for expressions is as follows (from the highest to the lowest)

1. Array referencing
2. Function calls & Pre-defined math functions
3. Multiplicative Operators
4. Additive Operators
5. Relational Operators

D.1. Array References

An array name followed by an index number enclosed by parenthesis is used to reference an integer array element. The

expression text(3) is an example of referencing the fourth element in an array “text”.

D.2. Function Calls

A function call is using a function designator followed by a colon and function arguments separated by commas. The entire function call is enclosed by square brackets.

[<function name>: <argument 1> <argument 2> ….];

Pre-defined math functions – sqrt, sum, avg, abs, and log are considered as unary operator.

D.3. Multiplicative Operators

The multiplicative operators are *, %, /, and ^ are the left associative.

D.4. Additive Operators

The additive operators + and – are the left associative.

D.5. Relational Operators

The relational operators ==, !=, <=, >=, >, and < are the left associative. If the comparison involving relational operators is true, it will return 1, otherwise will return 0.

D.6. Assignment Expression

The assignment (=) is the right associative.

D.7. Built in functions

Built in mathematical operations for this language has sqrt, sum, avg, abs, log. Their call convention is different from regular function calls.

<function name> [parameter];

E. Statements

E.1. Conditional Statement

The conditional expression is similar to C's if-else statements. It starts with if clause followed by a block of code enclosed by curly braces. Another block may be followed by else with another block of code enclosed by curly braces as an alternative.

```
if [<condition>] {
        <block>
}
else{
        <block>
}
```

E.2. Loops

For-loop and while-loop are supported. For while-loop, while keyword is followed by a condition. The condition statement is enclosed by square brackets. The loop keep runs until the condition is not true anymore.

```
while [<condition>]
{
        <block>
}
```

For-loop also runs until the condition is not true anymore. The syntax is slightly different.

```
for[<initialization>; <condition to run>; <increment>]{
        <block>
}
```

E.3. Return Statement

Return statements are required for functions with “int” or “array” return type. The returning expression’s type should match with the function’s return type.

E.4. Print

This language provide a built in output function to display expressions to standard output.

print[expression];

Project Plan

Process

The planning started when a project proposal was submitted to Professor Edwards. Based on his feedback, the proposal has been modified multiple times. After revisions, I formed concrete ideas on how the programming language would look like. Based on those plans, a scanner and a parser have been developed. The language reference manual was written based on syntax and flows written in scanner/parser.

While developing an abstract syntax tree and actual compiler, the language reference manual has been used as guidance. As I gained more knowledge on compiler concepts, I gradually modified the scanner and parser to accommodate the need of compiler code. MicroC compiler code has been used as a template for AAL, and that has been great help for this project. The target language for my compiler was Java code to be compiled by Java compiler.

The testing suite has been developed using a shell script. I wrote all test cases first before I develop new features. As I add new features, each test script helped to debug errors.

Programming Style

When writing parser, a single symbol could add many rules, but I tried to separate rules out with different symbols to make it easier to read and maintain. Some comments have been used to describe what the code does. I also tried to use intuitive variable and keyword names. I tried to follow Ocaml's conventional coding style in general.

Timeline & Log

- Feb 8th 2012 – Project Proposal submitted

- Feb 22nd 2012 – Project Proposal Revision based on Professor Edwards' feedback

- March 7th 2012 – The first version of scanner and parser was developed and compiled

- March 19th 2012 – The language reference manual was submitted.

- April 3rd 2012 – A simple hello world program was successfully compiled to JAVA

- April 11th 2012 – A test suite with 21 test cases have been developed

- April 28th 2012 – All features for AAL compiler have been developed, and tested.

- May 9th 2012 – The final project report is completed.

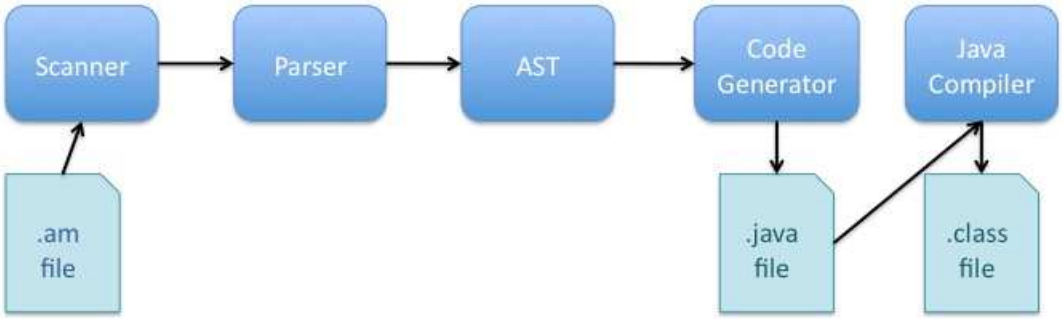
Development Environment

Eclipse Ocaml plug in was used to type code, but old-fashioned terminal was used to run program.

- OS : MacOS X 10.6.8 (Snow Leopard)
- Programming Language : Ocaml 3.12.0
- Text editor: Eclipse + OCamlIDE plug-in (editing only)
- Java : jdk 1.6.0

- Main test script : Bash Shell

Architectural Design



Front-end

For the convention, .am extension is used to indicate the file written in this language. The .am file is taken as an input, and the scanner scan tokens specified in scanner.mll. ocamllex is used to generate scanner.ml. ocamlyacc and parser.mly is used to parse to generate an abstract syntax tree. The abstract syntax tree is defined in AST.mli.

Back-end

Based on the abstract syntax tree created from the front-end, compile.ml is used to generate Java code from the tree. As Java requires the class name should be the same as file name. The name AAL is used to define class and file name. The java code can eventually be compiled using a Java compiler. In my case, I used jdk 1.6 to test. AMath.java file defines all built-in functions to be used in the generated Java code. When compiling the intermediate Java code, AMath.java file should also present in the same place.

Test Plan

A test suite was built to cover almost all basic aspects of this language. The MicroC's test suite was used as a model to build many simple test cases to cover single features rather than writing longer cases cover multiple features. Below are two sample programs with generated Java code.

Example 1.

Source code:

```
func array assignarray:int a,int b,int c,int d
{
  array g(4);
  g(0) = a;
  g(1) = b;
  g(2) = c;
  g(3) = d;
  return g;
}

func void execute:{
  array t(4);
  int i;
  t = [assignarray:32,19,183,7];
  for [i = 0 ; i < 4 ; i = i + 1] {
       print[t(i)];
  }
}
```

Generated code:

```
public class AAL {
public static void execute()
{
int[] t= new int[4];
int i;
t = AAL.assignarray(32, 19, 183, 7);
for (i = 0 ; i < 4 ; i = i + 1) {
System.out.println(t[i]);
}
```

```
}

public static int[] assignarray(int a, int b, int c, int d)
{
int[] g= new int[4];
g[0] = a;
g[1] = b;
g[2] = c;
g[3] = d;
return g;
}

public static void main(String args[]){AAL.execute();}}
```

Example 2.

Source Code:

```
func void execute:{
  int i;
  i = 1;
  if [i == 1] print[32]; else print[4];
  print[92];
}
```

Generated Code:

```
public class AAL {
public static void execute()
{
int i;
i = 1;
if (i == 1)
System.out.println(32);
else
System.out.println(4);
System.out.println(92);
}

public static void main(String args[]){AAL.execute();}}
```

Test suite & Automation

A Bash shell script (testall.sh) is used to execute all test cases. testall.sh from MicroC was used as a starting point. 23 test cases were written in .am extension files. Each .am file is compiled and AAL.java file is generated. A Java compiler is used to compile AAL.java for each case and make a comparison to corresponding .out file to verify the test is passed. Most of those scripts were written before completing compiler development, and they were used to verify new feature is successfully added.

Testcases were chosen based on contents in the language reference manual. I tried to add at least one test case for each title in the reference manual. Lastly, to run automation test suite, run make in the amath folder. After the build is complete, run "sh testall.sh ../amath/test/*.am". teatall.log display all related log messages.

testall.sh

```
#!/bin/sh

AMATH="./amath"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.am files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
    echo "FAILED"
    error=1
    fi
    echo "  $1"
```

```
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any,
written to difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
    SignalError "$1 differs"
    echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
    SignalError "$1 failed on $*"
    return 1
    }
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                            s/.am//'`
    reffile=`echo $1 | sed 's/.am$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2
    #generatedfiles="AAL.java AAL.class ${basename}.java.out"

    generatedfiles="$generatedfiles" &&
    Run "$AMATH" "<" $1 "> AAL.java" &&
    javac AAL.java &&
    java AAL > ${basename}.g.out &&
    Compare ${basename}.g.out ${reffile}.out ${basename}.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
    if [ $keep -eq 0 ] ; then
        rm -f $generatedfiles
      rm -f ${basename}.g.out
      rm AAL.java AAL.class
```

```
    fi
    echo "OK"
    echo "###### SUCCESS" 1>&2
    else
    echo "###### FAILED" 1>&2
    globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
    k) # Keep intermediate files
        keep=1
        ;;
    h) # Help
        Usage
        ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/fail-*.am tests/test-*.am"
fi

for file in $files
do
    case $file in
    *test-*)
        Check $file 2>> $globallog
        ;;
    *fail-*)
        CheckFail $file 2>> $globallog
        ;;
    *)
        echo "unknown file type $file"
        globalerror=1
        ;;
    esac
done

exit $globalerror
```

Test scripts

test-abs.am

```
func void execute:{
  int result;
  result = abs[-3];
  print[result];
}
```

test-arith1.am

```
func void execute:{
     print [3 + 2];
}
```

test-arith2.am

```
func void execute:{
     print [7 * 3 + 5 / 2 % 3];
}
```

test-array-var.am

```
func void execute:{
  array a(3);
  a(2) = 19;
  a(1) = -2;
  print[a(2)];
  print[a(1)];
}
```

test-avg.am

```
func void execute:{
  array t(3);
  int result;
  t(0) = 2;
  t(1) = 3;
  t(2) = 7;
  result = avg[t];
  print[result];
}
```

test-for1.am

```
func void execute:{
       int i;
```

```
  for [i = 0 ; i < 5 ; i = i + 1] {
    print[i];
  }
}
```

test-func1.am

```
func int add:int a, int b
{
  return a + b;
}

func void execute:{
  int a;
  a = [add:42, 3];
  print[a];
}
```

test-func2.am

```
func int calc:int x, int y
{
  return 0;
}

func void execute:{
  int i;
  i = 1;

  [calc: i = 2, i = i+1];

  print[i];
}
```

test-func3.am

```
func void printem:int a,int b,int c,int d
{
  print[a];
  print[b];
  print[c];
  print[d];
}

func void execute:{
  [printem:32,19,183,7];
}
```

test-func4.am

```
func array assignarray:int a,int b,int c,int d
{
  array g(4);
  g(0) = a;
  g(1) = b;
  g(2) = c;
  g(3) = d;
  return g;
}

func void execute:{
  array t(4);
  int i;
  t = [assignarray:32,19,183,7];
  for [i = 0 ; i < 4 ; i = i + 1] {
      print[t(i)];
  }
}
```

test-func5.am

```
func void assignarray:array a, int b
{
  a(b) = 4;
}

func void execute:{
  array t(3);
  [assignarray:t, 2];
  print[t(2)];
}
```

test-gcd.am

```
func int gcd:int a, int b{
  while [a != b] {
    if [a > b] a = a - b;
    else b = b - a;
  }
  return a;
}

func void execute:{
  print[gcd:2,14];
  print[gcd:3,15];
  print[gcd:99,121];
```

```
}
```

test-if1.am

```
func void execute:{
  int i;
  i = 1;
  if [i == 1] print[42];
  print[17];
}
```

test-if2.am

```
func void execute:{
  int i;
  i = 1;
  if [i == 1] print[32]; else print[4];
  print[92];
}
```

test-if3.am

```
func void execute:{
  int i;
  i = 1;
  if [i == 0] print[31];
  print[49];
}
```

test-if4.am

```
func void execute:{
  int i;
  i = 1;
  if [i == 0] print[32]; else print[4];
  print[92];
}
```

test-int-var.am

```
func void execute:{
  int a;
  a = 19;
  print[a];
}
```

test-log.am

```
func void execute:{
  int result;
  result = log[10^2];
  print[result];
}
```

test-ops1.am

```
func void execute:{
  print[1 + 2];
  print[1 - 2];
  print[1 * 2];
  print[100 / 2];
  print[99];
  print[1 == 2];
  print[1 == 1];
  print[99];
  print[1 != 2];
  print[1 != 1];
  print[99];
  print[1 < 2];
  print[2 < 1];
  print[99];
  print[1 <= 2];
  print[1 <= 1];
  print[2 <= 1];
  print[99];
  print[1 > 2];
  print[2 > 1];
  print[99];
  print[1 >= 2];
  print[1 >= 1];
  print[2 >= 1];
}
```

test-pow.am

```
func void execute:{
  int result;
  result = 2^4;
  print[result];
}
```

test-sqrt.am

```
func void execute:{
```

```
  int result;
  result = sqrt[4];
  print[result];
}
```

test-sum.am

```
func void execute:{
  array t(3);
  int result;
  t(0) = 2;
  t(1) = 5;
  t(2) = 9;
  result = sum[t];
  print[result];
}
```

test-while.am

```
func void execute:{
  int i;
  i = 9;
  while [i > 0] {
    print[i];
    i = i - 1;
  }
  print[99];
}
```

Lessons Learned

Through this course and project, I learned substantial amount of knowledge in programming language internals. I could see how source code gets processed through a compiler's different components, and I gained valuable skills to develop a simple compiler and a programming language. After such experiences, I feel that I can learn new languages quicker and deeper than before.

Initially, compiler concept was vague, and the idea of implementing a compiler was a daunting task for me. For that reason, I was reluctant to use Ocaml to implement a simple compiler. Implementing very simple features in Ocaml way of programming took a lot longer time, but elegant pattern matching style and expressive power were fitting very well for a compiler development task.

Balancing between class works and work has been particularly difficult this semester as I had too much pressures and deadlines coming from work. That was another good lesson I had. In my opinion, the dragon book is not for beginners. Reading through every word on that book could make students lost. I suggest future students to use the book as a reference purpose rather than as a guide. Another suggestion is to allocate extra time to fix Ocaml code's compiler errors. Some of them were very difficult to find, and it took days to find very small bug. Overall, it was a great learning experience for me.

Appendix (Code Listing)

scanner.ml

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"      { comment lexbuf }           (* Comments *)
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '/'       { DIVIDE }
| '%'       { MOD }
| '^'       { POWER }
| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| '['       { LBRACKET }
| ']'       { RBRACKET }
| ';'       { SEMI }
| ':'       { COLON }
| ','       { COMMA }
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "func"    { FUNC }
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }
| "return"  { RETURN }
| "int"     { INT }
```

```
| "void"   { VOID }
| "array"  { ARRAY }
| "sqrt"   { SQRT }
| "sum"    { SUM }
| "avg"    { AVG }
| "abs"    { ABS }
| "log"    { LOG }
| "print"  { PRINT }
| ['-']*['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm {
ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped
char)) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

parser.ml

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
COLON
%token PLUS MINUS TIMES DIVIDE MOD POWER ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token SQRT SUM AVG ABS LOG
%token RETURN IF ELSE FOR WHILE INT VOID ARRAY FUNC PRINT
%token <int> LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD POWER
%left SQRT SUM AVG ABS LOG

%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [] }
```

```
 | program functiondecl { $2 :: $1 }

functiondecl:
   functype ID COLON formals_opt LBRACE variabledecl_list
stmt_list RBRACE
     { { ftype = $1;
                 fname = $2;
       formals = $4;
       locals = List.rev $6;
       body = List.rev $7 } }

functype:
           FUNC VOID  { VoidRet }
      | FUNC INT { IntRet }
      | FUNC ARRAY { ArrayRet }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formaldecl:
           INT ID               { Int($2) }
      | ARRAY ID              { Array($2) }

formal_list:
    formaldecl                          { [$1] }
  | formal_list COMMA formaldecl   { $3 :: $1 }

variabledecl_list:
    /* nothing */    { [] }
  | variabledecl_list variabledecl { $2 :: $1 }

variabledecl:
    INT ID SEMI { IntVar($2) }
      | ARRAY ID LPAREN LITERAL RPAREN SEMI { ArrayVar($2, $4) }

variablecall:
           ID                { IntVarCall($1) }
      | ID LPAREN expr RPAREN { ArrayVarCall($1, $3) }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LBRACKET expr RBRACKET stmt %prec NOELSE { If($3, $5,
Block([])) }
  | IF LBRACKET expr RBRACKET stmt ELSE stmt    { If($3, $5, $7)
}
```

```
  | FOR LBRACKET expr_opt SEMI expr_opt SEMI expr_opt RBRACKET
stmt
     { For($3, $5, $7, $9) }
  | WHILE LBRACKET expr RBRACKET stmt { While($3, $5) }
     | PRINT expr SEMI { Print($2) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    LITERAL          { Literal($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
     | expr MOD    expr { Binop($1, Mod,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
     | expr POWER  expr { Power($1, $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
     | SQRT expr        { Uniop(Sqrt, $2) }
     | SUM expr         { Uniop(Sum, $2) }
     | AVG expr         { Uniop(Avg, $2) }
     | ABS expr         { Uniop(Abs, $2) }
     | LOG expr         { Uniop(Log, $2) }
     | variablecall     { Var($1) }
  | variablecall ASSIGN expr   { Assign($1, $3) }
  | LBRACKET ID COLON actuals_opt RBRACKET { Call($2, $4) }
  | LBRACKET expr RBRACKET { $2 }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

ast.mli

```
type op = Add | Sub | Mult | Mod | Div | Equal | Neq | Less | Leq
| Greater | Geq

type uop = Sqrt | Sum | Avg | Abs | Log

type variable =
          IntVar of string
```

```
      | ArrayVar of string * int

type fparam =
           Int of string
      | Array of string

type varcall =
           IntVarCall of string
      | ArrayVarCall of string * expr
and expr =
    Literal of int
  | Var of varcall
  | Binop of expr * op * expr
      | Uniop of uop * expr
      | Power of expr * expr
  | Assign of varcall * expr
  | Call of string * expr list
  | Noexpr

type functiontype =
           VoidRet
      | IntRet
      | ArrayRet

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
      | Print of expr

type functiondecl = {
           ftype : functiontype;
    fname : string;
    formals : fparam list;
    locals : variable list;
    body : stmt list;
  }

type program = functiondecl list
```

compile.ml

```
open Ast

let string_of_variable = function
           ArrayVar(s, i) -> "int[] " ^ s ^ "= new int[" ^
string_of_int i ^ "];\n"
      | IntVar(s) -> "int " ^ s ^ ";\n"
```

```
let rec string_of_variablecall = function
        IntVarCall(s) -> s
      | ArrayVarCall(s, e) -> s ^ "[" ^ string_of_expr e ^ "]"
and string_of_expr = function
    Literal(l) -> string_of_int l
      | Var(s) -> string_of_variablecall s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
      Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/" | Mod ->
"%"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=")
^ " " ^
      string_of_expr e2
      | Uniop(o, e) -> "AMath." ^ (match o with
          Sqrt -> "sqrt("
      | Sum -> "sum("
      | Avg -> "avg("
      | Abs -> "abs("
      | Log -> "log(" ) ^
          string_of_expr e ^ ")"
  | Assign(v, e) -> string_of_variablecall v ^ " = " ^
string_of_expr e
      | Power(e1, e2) -> "AMath.pow(" ^ string_of_expr e1 ^ ", "
^ string_of_expr e2 ^ ")"
  | Call(f, el) ->
           "AAL." ^ f ^ "(" ^ String.concat ", " (List.map
string_of_expr el) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^
"}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^
" ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
string_of_stmt s
      | Print(e) -> "System.out.println(" ^ string_of_expr e ^
");\n"

let string_of_fparam = function
```

```
    Int(id) -> "int " ^ id
  | Array(id) -> "int[] "^ id

let string_of_functiontype = function
          VoidRet -> "void"
      | IntRet -> "int"
      | ArrayRet -> "int[]"

let string_of_fdecl fdecl =
  "public static " ^ string_of_functiontype fdecl.ftype ^ " " ^
     fdecl.fname ^ "(" ^ String.concat ", " (List.map
string_of_fparam fdecl.formals) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_variable fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program funcs =
      "public class AAL { \n" ^
  String.concat "\n" (List.map string_of_fdecl funcs) ^ "\n" ^
      "public static void main(String args[]){"^
      "AAL.execute();"^
      "}"^
      "} \n"
```

amath.ml

```
let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
     let listing = Compile.string_of_program program in
print_string listing
```

Makefile

```
OBJS = parser.cmo scanner.cmo compile.cmo amath.cmo

TESTS = \
abs \
arith1 \
arith2 \
array-var \
for1 \
func1 \
func2 \
func3 \
func4 \
func5 \
gcd \
if1 \
if2 \
```

```
if3 \
if4 \
int-var \
log \
ops1 \
pow \
sqrt \
sum \
while

TARFILES = Makefile scanner.mll parser.mly ast.mli compile.ml
amath.ml testall.sh \
     $(TESTS:%=tests/test-%.am) \
     $(TESTS:%=tests/test-%.out)

amath : $(OBJS)
     ocamlc -o amath $(OBJS)

.PHONY : test
test : amath testall.sh
     ./testall.sh

scanner.ml : scanner.mll
     ocamllex scanner.mll

parser.ml parser.mli : parser.mly
     ocamlyacc parser.mly

%.cmo : %.ml
     ocamlc -c $<

%.cmi : %.mli
     ocamlc -c $<

amath.tar.gz : $(TARFILES)
     cd .. && tar zcf amath/amath.tar.gz $(TARFILES:%=amath/%)

.PHONY : clean
clean :
     rm -f amath parser.ml parser.mli scanner.ml *.cmo *.cmi
*.out *.diff

# Generated by ocamldep *.ml *.mli
amath.cmo: scanner.cmo parser.cmi ast.cmi
amath.cmx: scanner.cmx parser.cmx ast.cmi
compile.cmo: scanner.cmo parser.cmi ast.cmi
compile.cmx: scanner.cmx parser.cmx ast.cmi
parser.cmo: ast.cmi parser.cmi
parser.cmx: ast.cmi parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmi
```

AMath.java

```
public class AMath{
	public static int sqrt (int num){
		return (int) Math.sqrt(num);
	}

	public static int sum(int[] nums){
		int s = 0;
		for(int i=0; i<nums.length; i++) s+=nums[i];
		return s;
	}

	public static int avg(int[] nums){
		return AMath.sum(nums)/nums.length;
	}

	public static int abs(int num){
		return Math.abs(num);
	}

	public static int pow(int num1, int num2){
		return (int) Math.pow(num1, num2);
	}

	public static int log(int num){
		return (int) Math.log(num);
	}
}
```