COMS W4115
Spring 2012

Advanced Arithmetic Language

Language Reference Manual

UNI: jc3783
Name: Jimin Choi

A. Introduction

The manual describes the Advanced Arithmetic Language supporting basic and advanced mathematical operations such as square roots, powers, and summations. It also supports conventional programming language features - if-else statement blocks, loops, and variable declarations. This language is intended for a class compiler implementation project for COMS W4115.

B. Lexical Conventions

B.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.

B.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*
- *power*
- *sum*
- *avg*

B.3. Constants

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

B.4. Comments

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

C. Meaning of Identifiers

In this language, identifiers indicate functions or variables.

C.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 When the function returns, the argument name is no longer valid.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.

C. 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> = <constant>

For array variable, the same assignment operator is used to assign another array to the array variable, or curly braces can be used to specify a sequence of integers in array.

D. Data Types

D.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.

D.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 square brackets enclosing the index number.

D.3 Void

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

E. 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

E.1. Array References

An array name followed by an index number enclosed by square brackets is used to reference an integer array element. The expression text[3] is an example of referencing the fourth element in an array “text”.

E.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 are considered as unary operator.

F.3. Multiplicative Operators

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

F.4. Additive Operators

The additive operators + and – are the left associative.

F.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.

F.6 Assignment Expression

The assignment (=) is the right associative.

F.7. Conditional Expression

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>
}
```

F.8. 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>
}
```

G. Scanner and Parser

scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'      { MOD }
| '('      { 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 }
| "power"   { POWER }
| "sum"     { SUM }
| "avg"     { AVG }
| ['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.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA
COLON
%token PLUS MINUS TIMES DIVIDE MOD ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token SQRT POWER SUM AVG
%token RETURN IF ELSE FOR WHILE INT VOID ARRAY FUNC
%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
%left SQRT POWER SUM AVG

%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [] }
 | program variabledecl { ($2 :: fst $1), snd $1 }
 | program functiondecl { fst $1, ($2 :: snd $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  { Void }
      | FUNC INT { Int }
      | FUNC ARRAY { Array }

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

formal_list:
    INT ID               { [Int($2)] }
      | ARRAY ID               { [Array($2)] }
  | formal_list COMMA INT ID { Int($4) :: $1 }
      | formal_list COMMA ARRAY ID { Array($4) :: $1 }

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

variabledecl:
   INT ID SEMI { $2 }

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) }

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

```
expr:
    LITERAL          { Literal($1) }
  | ID               { Id($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 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) }
      | expr POWER expr  { Binop($1, Power, $3) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | LBRACKET ID COLON actuals_opt RBRACKET { Call($2, $4) }
  | LPAREN expr RPAREN { $2 }

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

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