Arithmetic Calculation Language
Language Reference

Nathan Corvino

March 21, 2011

1 Introduction

This document describes the Arithmetic Calculation Language (ACL), a small subset of C that allows numeric colculations to be done in a procedural fashion. The language only allows integer values to be manipultated, hence the name Arithmetic Calculation Language. It does, however, allow for arrays of integers to be declared, and provides a language features that are illustrative of the core procedural constructs: variable declarations, function definitions, selection, and iteration.

2 Lexcial Conventions

A program consists of a single file, containing global variable declarations and procedure definitions. Execution begins with the function named main, which is required.

2.1 Comments

Comments are introduced by /* or //. Comments introduced by /* are terminated with */, while comments introduced by // terminate at the next newline.

2.2 Identifiers

Identifiers consist of letters, digits, and underscores, and must begin with a letter or underscore; case is significant.

2.3 Keywords

The following identifiers are used as keywords, and may not be used as identifiers for functions or variables:

- if
- else
- while
- int
- void

2.4 Constants

Integer constants consist of a series of digits, and are treated as decimal numbers.

3 Variables Declarations

Variables are introduced by their type–the only allowed type is int–folowed by an identifier. If the identifier is followed by an expression in square brackets— [] —then the variable is treated as an array–a block of contiguous storage ints represented by the expression value. If the identifier appears without brackets, it stores a single integer value.

```
variable_declaration:
    int ID
  | int ID[expression]
```

Assignment to identifiers is done an expression in the form E1 = E2, where E1 must be a non-array identifier, or an array identifier with an index expression in brackets. The expression E1 = E2 also has the value of E2.

Variable declarations can appear outside of all function declarations, in which case they are available throught the program; or they appear at the beginning of a block–that is, in a list of statements enclosed in curly-braces, before the first statement.

4 Function Definitions

Functions are defined by a return type—either int or void, arrays are not supported—an identifier that names the function, a list of parameters separated by commas, enclosed in parantheses, and a block of code enclosed in curly braces. Parameters are optional, but the paranetheses are mandatory.

```
function_return_type:
    int
  | void

function_definition:
    function_return_type ID LEFT_PAREN parameter_opt RIGHT_PAREN compound_statement
```

```
parameter_opt:
    /* nothing */
  | parameter_list

parameter_list:
    parameter_declaration
  | parameter_list, parameter_declaration

parameter_declaration:
    int ID
```

The parameters named are available as identifiers within the block of code; when the function is called, these parameters are matched with expressions in the calling code–the function call arguments–to supply their values.

5 Expressions

Expression precedence is the same as the order of the following subsections, highest order first; the associativity of each operator is specified within the subsection.

```
expression:
  | ID(optional argument list)
  | expression * expression
  | expression / expression
  | expression + expression
  | expression - expression
  | expression < expression
  | expression > expression
  | expression <= expression
  | expression >= expression
  | expression = expression
  | expression != expression
  | variable_reference = expression
```

5.1 Function Calls

Function calls consist of an identifier followed by parantheses which contain a comma separated list of arguments to the function call. The identifier and arguments must match with a declared function, and the type can either be int or void; void indicates that the function call cannot be contained within other expressions, as it does not return a value.

When calling a function, a copy of each argument is made, and assigned to corresponding parameter in the function definition; the number of arguments must match with the previous declaration. The block of the function declaration is then exectued with the copied values, with the expression evaluting to the value returned, if the function returns an int.

5.2 Multiplictive Operators

Multiplication and division are binary operators specified by * and /, respectively. They associate left-to-right.

5.3 Additive Operators

Addition and subtraction are binary operators specified by + and -, respectively. They associate left-to-right.

5.4 Relational Operators

The relational operators perform numeric comparisons: <, >, <=, >=. They associate left-to-right, and return one of the specified comparison is true, 0 if it is false.

5.5 Equality Operators

Equality comparisons test for numeric equality, == test for equality, and != tests for inequality. They associate left-to-right, and return 1 if the relation holds, 0 if it does not.

5.6 Assignment

The assignment operation, =, groups right-to-left. It requires a declared identifier as its left operator (lvalue), and that identifier must not have been declared an array, or must have an index expression contained in brackets. In the expressions E1 = E2, E2 is stored into the lvaue E1, and the value of the expression is E2.

6 Statements

In addition to expresions, there are statements for selection—if; iteration—while; and the statement to return a value form a function. In addition, curly-braces can contain a list of variable declarations, followed by a list of statements, is a statement.

```
statement:
    expression;
  | RETURN expression;
  | compound_statement
  | if (expression) statement
  | if (expression) statement else statement
  | while (expression) statement

compound_statement:
    { declaration_opt statement_opt }

declaration_opt:
    /* nothing */
  | declaration_list

declaration_list:
    /* nothing */
  | declaration_list variable_declaration
```

6.1 if Statement

The if statement performs selection, and takes the form:

```
  if (expression) statement
| if (expression) statement else statement
```

The else statement binds with the closest if statement. If the expression evaluates to 0, the else statement is evaluated if present; if the expression evaluates to anything else, then the if statement is executed.

6.2 while Statement

The while statement performs iteraion, and takes the form:

```
while (expression) statement
```

The while statement evaluates the specified expression, and if it evalutes to 0, skips its specified statement and proceeds to the subsequent statement. If the specified statement evaluates to anything but 0 then the specified statement is executed and this process is repeated; that is, the expression is evaluated again, and either the while statement is finished, or the process continues.

6.3 return Statement

From within a function, the return statement ends execution of the function block, returning to the calling code; the specified statement provides the return value. The return statement is required in a function that declares itself to return an int.

7 Compiler Front End

7.1 scaner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }
| "/*"                 { comment lexbuf }

| ";"                  { SEMICOLON }
| ","                  { COMMA }

| "("                  { LEFT_PAREN }
| ")"                  { RIGHT_PAREN }
| "{"                  { LEFT_BRACE }
| "}"                  { RIGHT_BRACE }
| "["                  { LEFT_BRACKET }
| "]"                  { RIGHT_BRACKET }

| "+"                  { PLUS }
| "-"                  { MINUS }
| "*"                  { TIMES }
| "/"                  { DIVIDE }
| "="                  { ASSIGN }

| "=="                 { EQUAL }
| "!="                 { NOT_EQUAL }
| "<"                  { LESS_THAN }
| ">"                  { GREATER_THAN }
| "<="                 { LESS_EQUAL }
| ">="                 { GREATER_EQUAL }

| "if"                 { IF }
| "else"               { ELSE }
| "while"              { WHILE }

| "int"                { INT }
| "void"               { VOID }

| eof                  { EOF }

| ['0'-'9']+ as lit    { LITERAL(int_of_string lit) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lit  { ID(lit) }

| _ as char     { raise (Failure("illegal character " ^ Char.escaped char)) }

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

and line_comment = parse
  ['\r' '\n']  { token lexbuf }
```

```
| _           { comment lexbuf }
```

7.2 ast.ml

```
type operator = Add | Sub | Mult | Div | Equal | NotEqual | LessThan | GreaterThan
        | LessEqual | GreaterEqual

type return_type =
    Int
  | Void

type variable = string * string

type variable_ref =
    Id of string
  | Array of string * expression
and expression =
    Literal of int
  | VarRef of variable_ref
  | Binop of expression * operator * expression
  | Assign of variable_ref * expression
  | Call of string * expression list
  | Noexpr

type statement =
    Block of variable list * statement list
  | Expression of expression
  | Return of expression
  | If of expression * statement * statement
  | While of expression * statement

type function_definition = {
    fname : string;
    formals : string list;
    body : statement list
  }

type program = variable list * function_definition list
```

8 parser.mly

```
%{ open Ast %}

%token SEMICOLON COMMA
%token LEFT_PAREN RIGHT_PAREN LEFT_BRACE RIGHT_BRACE LEFT_BRACKET RIGHT_BRACKET
```

```
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQUAL NOT_EQUAL LESS_THAN GREATER_THAN LESS_EQUAL GREATER_EQUAL
%token RETURN IF ELSE WHILE INT VOID
%token <int> LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left EQUAL NOT_EQUAL
%left LESS_THAN GREATER_THAN LESS_EQUAL GREATER_EQUAL
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

program:
    /* nothing */     { [], [] }
  | program variable_declaration  { ($2 :: fst $1), snd $1 }
  | program function_definition   { fst $1, ($2 :: snd $1) }

function_return_type:
    INT    { INT }
  | VOID   { VOID }

function_definition:
    function_return_type ID LEFT_PAREN parameter_opt RIGHT_PAREN compound_statement
      { {
        fname = $2;
        formals = $4;
        body = $6;
      } }

parameter_opt:
    /* nothing */    { [] }
  | parameter_list   { List.rev $1 }

parameter_list:
    parameter_declaration                        { $1 }
  | parameter_list COMMA parameter_declaration   { $3 :: $1 }

parameter_declaration:
    INT ID    { Id($2) }

variable_declaration:
    INT ID    { Id($2) }
  | INT ID LEFT_BRACE expression RIGHT_BRACE   { Array($2, $4) }

variable_reference:
    ID                                      { Id($1) }
  | ID LEFT_BRACKET expression RIGHT_BRACKET { Array($1, $3) }

compound_statement:
    LEFT_BRACE declaration_opt statement_opt RIGHT_BRACE   { Block($2, $3) }

declaration_opt:
    /* nothing */     { [] }
  | declaration_list  { List.rev $1 }

declaration_list:
    /* nothing */                             { [] }
  | declaration_list variable_declaration   { $2 :: $1 }

statement_opt:
    /* nothing */   { [] }
  | statement_list  { List.rev $1 }

statement_list:
    /* nothing */             { [] }
  | statement_list statement  { $2 :: $1 }

statement:
    expression SEMICOLON { Expr($1) }
  | RETURN expression SEMICOLON { Return($2) }
  | compound_statement { $1 }
  | IF LEFT_PAREN expression RIGHT_PAREN statement %prec NOELSE { IF($3, $5, Block([], [])) }
  | IF LEFT_PAREN expression RIGHT_PAREN statement ELSE statement { IF($3, $5, $7) }
  | WHILE LEFT_PAREN expression RIGHT_PAREN statement { While($3, $5) }

expression:
    LITERAL                    { Literal($1) }
  | variable_reference         { VarRef($1) }
  | expression PLUS expression { Binop($1, Add, $3) }
  | expression MINUS expression { Binop($1, Sub, $3) }
  | expression TIMES expression { Binop($1, Mult, $3) }
  | expression DIVIDE expression { Binop($1, Div, $3) }
  | expression EQUAL expression { Binop($1, Equal, $3) }
  | expression NOT_EQUAL expression { Binop($1, NotEqual, $3) }
  | expression LESS_THAN expression { Binop($1, LessThan, $3) }
  | expression GREATER_THAN expression { Binop($1, GreaterThan, $3) }
  | expression LESS_EQUAL expression { Binop($1, LessEqual, $3) }
  | expression GREATER_EQUAL expression { Binop($1, GreaterEqual, $3) }
  | variable_reference ASSIGN expression { Assign($1, $3) }
  | ID LEFT_PAREN argument_opt RIGHT_PAREN { Call($1, $3) }
  | LEFT_PAREN expression RIGHT_PAREN { $2 }

argument_opt:
    /* nothing */  { [] }
  | argument_list  { List.rev $1 }

argument_list:
    expression                      { [$1] }
  | argument_list COMMA expression { $3 :: $1 }
```