Polynomial Calculator Programming Language (PCPL)
Reference Manual

Donghui Xu (dx2116)
COMS 4115 Project
Professor: Stephen Edwards

1. Introduction

PCPL is a simple programming language base on C programming language and the Microc language. It is designed to facilitate simple polynomial operations, such as addition, subtraction. It also allows programmers to write their own functions for more complex polynomial operations, such as multiplication, division etc.
This reference manual closely follows the C reference manual by Dennis M. Ritchie

2. Lexical conventions

2.1. Tokens

There are five types of tokens in PCPL, namely identifiers, keywords, constants, operators and other separators.

2.2. Comments

A comment starts with “/*” and ends with “*/”. The language does not support nested comments. Comments as well as white spaces including blanks, tabs and new lines are ignored.

2.3. Identifiers

An identifier consists of alphabetic characters and digits. An identifier must start with an alphabetic character and followed by zero or more alphabetic characters or digits. Identifiers are case sensitive. Variable names and function names are identifiers.

2.4. Keywords

Keywords are reserved identifiers. The following is the list of key words in this language:

```
if
else
for
while
return
int
float
polynomial
```

2.5. Constants

PCPL defines three types of constants, integers, floating point numbers and polynomials.

2.5.1. Integers

An integer is a sequence of one or more digits (0-9).

2.5.2. Floating point numbers

A floating point number consists of an integer part, a decimal point (.) and a fraction part. It is a sequence of digits (0-9) followed by a decimal point (.) followed by zero or more digits. Either the integer part or the fraction part (not both) may be missing.

2.5.3. Polynomials

A polynomial is represented by a list of ordered pairs. For example $5x^2 + 3x + 2$ is represented as [(5,2), (3,1),(2,0)]. Each element in the list represents a term in a polynomial expression. The first element in the two-element pair represents the coefficient of the term and the second element represents the exponent of the corresponding term.

3. Expressions

The precedence levels of these expressions are from higher to lower.

3.1. Primary expression

Primary expressions are identifiers, constants, parenthesized expressions and function calls group left-to-right. Primary expressions appear on the right hand side of the statement.

3.2. Unary operator

Unary operator (-) negates the resulting value of an expression group right-to-left. The expression could be an integer or a floating point number or a polynomial.

3.3. Multiplicative operators

The multiplicative operators are * and / group left-to-right.

multiplication: expr * expr

division: expr / expr

A constant of polynomial type can be the left operand of the division operator but not the right operand. Multiplicative operators have lower precedence than unary operator.

3.4. Additive operators

The additive operators are + and – group left to right.

addition : expr + expr
subtraction : expr – expr

Both operations are grouped left- to- right. Additive operators have lower precedence than multiplicative operators.

3.5. Relational operators

Relational operators are >, >=, <,<= group left-to-right.

greater than : expr > expr
greater or equal to: expr >= expr
less than : expr < expr
less than or equal to : expr <= expr

The operation return 0 if specified relation is false, returns 1 if the result is true. int, float and polynomial can be one either side of the operator. The operations will be evaluated according to conventional mathematical formula. Relational operators have lower precedence than additive operators.

3.6. Equality Operators

Equality Operators are ==, ! =.

equals to: expr == expr
not equals to: expr != expr

The operation return 0 if the specified relation is false, returns 1 if the result is true. int, float and polynomial can be on either side of the operator. Equality operators have the same preference as relational operators.

3.7. Assignment operator

Assignment operator is = group right-to-left.
lvalue = expr
In an assignment operation, the left operand is a lvalue and it gets assigned the value of the assignment expression. Both operands need to be the same type. Assignment operator has lower precedence than relational operators and equality operators.

4. Declarations

A variable needs to be declare before it can be assigned a value. Declaration starts with a type name, such as int, float or polynomial and followed by a list of identifiers.

int id1, id2;

5. Statements

Unless indicated statements are executed in sequence.

5.1. Expression statements

Most statements are expression statements. They have the form

```
expression;
```

Most expression statements are assignments or function calls.

5.2. Conditional statement

Conditional statements are if or if-else statements. They have the forms

```
if (expression){
statement list 1}

if (expression){
  statement list1}
else{
statement list 2
}
```

If expression is evaluated to nonzero the statement list 1 is executed; if 0 the statement list 2 is executed. To avoid else ambiguity one could connect an else with the last else-less if.

5.3. Iterative statement

Iterative statement has the form

```
while (expression)
{
  statement list
}
```

If the expression is evaluated to nonzero the statement list is executed. After the execution of the statement list the expression will be evaluated again and the statement list will be executed as long as the expression remains nonzero.

5.4. For statement

The for statement has the form

```
for ( expr1; expr2; expr3)
{
 statement list
}
```

The for statement is equivalent to

```
expr1;
```

```
while (expr2)
{
statement list
expr3
}
```

6. Functions

A function contains a sequence of statements. Functions have the form

```
return_type  function_name ( parameter list)
{
  statement list
  return expr;
}
```

The type of the expression in return statement must be the same as the return_type in the function declaration.

7. Program

A program consists of a main function, within the main function other functions can be called. Nesting functions are not allowed. An example of a program could be

```
main ()
{
   declarations;
   statements;
   function1();
  }
 function1()
{
}
```

Scanner.mll

```
{ open Parser }

let digit = ['0'-'9']

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf }    (* Whitespace *)
| "/*"     { comment lexbuf }          (* Comments *)
| '['          { LBRACKET }
| ']'      { RBRACKET }
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "float"  { FLOAT }
| "polynomial"   {POLYNOMIAL}
| digit+ as lxm  { LITINT(int_of_string lxm)
| "." digit+ | digit+ "." digit* as lxm { LITFLOAT(float_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 LBRACKET RBRACKET LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE FLOAT POLYNOMIAL
%token  <int> LITINT <float> LITFLOAT
%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

%start program
%type <Ast.program> program

%%

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

fdecl:
  ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
    { { fname = $1;
        formals = $3;
        locals = List.rev $6;
        body = List.rev $7 } }
```

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

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

vdecl_list:
   /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
  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 LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
    { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

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

term:
    LPAREN expr COMMA expr RPAREN { [($2,$4)] }
  |term COMMA LPAREN expr COMMA expr RPAREN { [($4 , $6)] :: $1 }

expr:
   LITINT        { Litint($1) }
  | LITFLOAT       { Litfloat($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 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) }
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }
  | LBRACKET term RBRACKET { List.rev $2 }

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

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

Ast.ml

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

type expr =
    Litint of int
  | Litfloat of float
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Pexpr of polynomial
  | Noexpr

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

type func_decl = {
    fname : string;
    formals : string list;
    locals : string list;
    body : stmt list;
  }

type program = string list * func_decl list

let rec string_of_expr = function
    Litint(l) -> string_of_int l
  | Litfloat (l) -> string_of_float l
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
       Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      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

let string_of_vdecl id = "int " ^ id ^ ";\n"
```

```
let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```