# Josh Reference Manual

Bora Elci, Burcu Cetin, Angel Garcia, Vasilis Benopoulos, Gregory Schare

# 1 Introduction

Josh is your friendly neighborhood shell scripting language. It is consistent with UNIX commands allowing to include bash scripts and enhanced with Python-like syntax to facilitate the learning of scripting and automation. Josh uses error and type checking prior to running scripts in order to allow for safety and avoid cluttering state if a part of the script is not valid. Unlike Perl and Bash, Josh is less complex and built for readability and reusability through intuitive syntax and statically scoped. Josh also differs from Python as it is more minimalistic and less generalized/abstract. Josh also aims to cut down running into runtime errors as it is a static, strongly typed language.

Josh is not intended for general-purpose use or for complex systems. Josh focuses on education, and it is designed for beginner to intermediate level programmers to familiarize themselves with bash, scripting languages, and UNIX command-line through readable code with easier syntax while minimizing error and expediting automation for routine tasks and testing.

# 2 Lexical Analysis and Conventions

## 2.1 Tokens

Josh has 5 types of tokens: identifiers, keywords, literals, operators, and special symbols (such as separators).

Whitespace (spaces, tabs, newlines) are ignored and have no meaning, except when they separate tokens.

Regular expression:

```
let whitespace = [' ' '\t' '\r' '\n']

rule token = parse
  | whitespace { token lexbuf }
```

## 2.2 Comments

Josh supports single line and block comments which start with the character sequence `{|` and end with `|}`. Comments are discarded by the scanner.

Comments may not be nested.

Example:

```
{| This is a single line comment. |}
```

```
{| This is a multi-line comment.
 | We recommend formatting comments with pipes like this.
 | The pipes make it easy to read.
 | (and we think it looks quite nice!)
 |}
```

Scanner rule:

```
rule token = parse
  | "{|"     { comment lexbuf }

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

## 2.3 Identifiers

Identifiers are a sequence of letters, digits, and underscores where the first character is a letter or an underscore. Identifiers cannot be keywords.

Example:
`foo`, `foo123`, `a`, `prevNode`, `snake_case_var`, `MdbRec`

Regular expression:

```
let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let id = (letter | '_') (letter | digit | '_')*
```

## 2.4 Keywords

The identifiers below are keywords which are reserved and can only be used for their specific purpose in the programming language.

| | | |
|---|---|---|
| and | while | string |
| or | break | void |
| not | continue | char |
| if | return | true |
| else | int | false |
| for | bool | record |
| in | float | |

`int`, `float`, `bool`, `char`, `void`, and `string` are used to declare the type of variables, fields of records, and arguments and return values of functions.

`true` and `false` are literal values for the `bool` type.

`and`, `or`, and `not` are logical operators.

`if`, `else`, `for`, `in`, `while`, `break`, `continue`, and `return` are associated with control flow statements.

`record` initiates the declaration of a new record type.

## 2.5 Literals

Josh supports literal values for five types: int, float, bool, char, and string.

For more details on how they are tokenized, see the Appendix section on the scanner.

### 2.5.1 int

Integer literals are a sequence of one or more digits in decimal, which may be preceded by a minus sign indicating it is negative.

Example:
`0`, `25`, `-1`, `12345`

Regular expression:

```
let digit = ['0'-'9']
let int = ('-')? digit+
```

### 2.5.2 float

Float literals are one or more digits with a decimal point somewhere before, after, or in the middle. Like integers, they may be negated.

Example:
`0.0`, `.5`, `12.34`, `1.000001`, `25.`

Regular expression:

```
let digit = ['0'-'9']
let float = ('-')? ((digit+) ['.'] digit*) | ((digit*) ['.'] digit+)
```

### 2.5.3 bool

Boolean literals have two values represented by the keywords `true` and `false`.

### 2.5.4 char

Character literals are ASCII characters enclosed in single quotes. Escape characters are also character literals starting with a backslash and followed by the specific ASCII characters listed below.

Example: `'a'`, `'5'`, `'\n'`, `'}'`, `' '`

Regular expression:

```
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let escape_char = ''' (escape) '''
```

```
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let char = ''' ( ascii ) '''
```

### 2.5.5 string

String literals are a sequence of zero or more characters enclosed in double quotes.

Example:
`"Hello, World!", "abc123", "", "    \n    ", "\\LaTeX"`

Regular expression:

```
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let string = '"' ( (ascii | escape)* as lem ) '"'
```

## 2.6 Operators

The operators can be grouped into four categories: arithmetic, assignment, comparison, and logical.

They are all scanned directly as strings or characters.

Regular expression:

```
rule token = parse
  (* Arithmetic *)
  | '+'      { PLUS }
  | '-'      { MINUS }
  | '*'      { MULT }
  | '/'      { DIV }
  | '%'      { MOD }
  (* Assignment *)
  | '='      { ASSIGN }
  (* Comparison *)
  | "=="     { EQ }
  | "!="     { NEQ }
  | '<'      { LT }
  | "<="     { LEQ }
  | '>'      { GT }
  | ">="     { GEQ }
```

```
  (* Logical *)
  | "and"    { AND }
  | "or"     { OR }
  | "not"    { NOT }
```

### 2.7 Separators

The following special symbols or separators are specific tokens reserved for separating other types of tokens.

- Commas delimit arguments and record fields.
- Semicolons delimit statements.
- Parentheses are used to group expressions and call functions.
- Braces are used to group blocks and construct records.
- Brackets are used to construct and index lists.
- Dots are used to access record fields and write literal floats.

Regular expression:

```
rule token = parse
  | ','      { COMMA }
  | ';'      { SEMI }
  | '('      { LPAREN }
  | ')'      { RPAREN }
  | '{'      { LBRACE }
  | '}'      { RBRACE }
  | '['      { LBRACK }
  | ']'      { RBRACK }
  | '.'      { DOT }
```

## 3 Types

Josh is strongly typed. Every expression has a type that is determined at compile time. Every variable must be declared with a type. When a function or operation is called with the wrong types, it will result in a type error.

Types can be split into two categories: primitive and non-primitive. ## Primitive Data Types The primitive data types are, as you might expect,

the same as those which have literal values described above. They are int, float, bool, char, and string. There is also void.

### 3.0.1 int

The int type stores integer values.

Example:

```
int age = 21;
```

Grammar:

```
typ:
  INT       { Int  }
```

### 3.0.2 float

The float type stores decimal values up to a certain precision.

Example:

```
float pi = 3.14159;
```

Grammar:

```
typ:
  FLOAT   { Float }
```

### 3.0.3 bool

The bool type stores true and false values.

Example:

```
bool isEmpty = false;
```

Grammar:

```
typ:
  BOOL    { Bool }
```

### 3.0.4 char

The char type stores single character values.

Example:

```
char letter = 'a';
```

Grammar:

```
typ:
  CHAR    { Char }
```

### 3.0.5 string

The string type stores a sequence of characters.

Example:

```
string name = "Josh";
```

Grammar:

```
typ:
  STRING  { String }
```

### 3.0.6 void

The void type does not store any value. It is used for functions that do not return anything.

Example:

```
void printA(string a) { echo(a); }
```

Grammar:

```
typ:
  VOID    { Void }
```

## 3.1 Non-Primitive Data Types

### 3.1.1 Lists

Josh also supports dynamic arrays, which we call lists. Every element of a list must be the same type. Lists can be nested.

Example:

```
[int] myList = [1, 2, 3];
[[char]] dnaPairs = [['a', 't'], ['c', 'g']];
```

Grammar:

```
typ:
  LBRACK typ RBRACK { ListT($2) }
```

### 3.1.2 Records

Records are similar to structs in C. A record creates a custom data type that can be used to associate a finite number of items of different types.

Example:

```
record Person = {
    string name,
    int age
}
Person josh = Person { "Josh", 21 };
```

Grammar:

```
typ:
  ID { RecordType($1) }
```

### 3.1.3 Functions

Functions can be types in Josh. This allows the programmer to include functions inside of lists and records in order to create powerful abstractions and generic algorithms.

Example:

```
record Node {
    int data;
```

```
    int cmp(Node n1, Node n2);
}
```

Grammar:

```
typ:
  typ ID LPAREN opts_list RPAREN { FuncType($2, $4, $1) }
```

### 3.2 Type conversions

Some types may be, in some cases, sensibly converted into another. All type conversions are explicit. See Standard Library.

## 4 Program

A Josh program consists of a series of statements and function declarations.

Grammar:

```
program:
  top_level_list EOF { $1 }

top_level_list:
  { [] }
  | top_level top_level_list { $1::$2 }

top_level:
    stmt { Stmt $1 }
  | fdecl { Fdecl $1 }
```

## 5 Statements

Statements are where the action happens. Whereas function declarations define how data can be transformed, statements define what kind of data there can be and actually perform these transformations. Most statements can be thought of as the “verbs” of your program.

## 5.1 Expressions

The simplest kind of statement is an expression followed by a semicolon. See Expressions.

Example:

```
x = 5;
```

Grammar:

```
stmt:
  expr SEMI  { Expr $1 }
```

## 5.2 Variable Declarations

Variable declarations count as statements when followed by a semicolon. See Variable Declarations.

Example:

```
int x;
int y = 5;
```

Grammar:

```
stmt:
  vdecl SEMI   { Vdecl $1 }
```

## 5.3 Blocks

Series of statements can be grouped into blocks using braces. They are read and executed sequentially.

Example:

```
{
    int x = 5;
    echo("Hello, World!");
}
```

Grammar:

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

stmt:
  LBRACE stmt_list RBRACE   { Block $2 }
```

## 5.4 Conditionals

Josh supports conditional operators identically to C.

The `if` keyword initiates a statement containing a condition, which is an expression with a Boolean value, and another statement (most commonly a series of statements surrounded by braces). Optionally, you can use the `else` keyword after the statement following the `if` to begin a statement or series of statements which are executed if the `if` condition is false.

We resolve the dangling else problem by using a `%prec` directive in our parser grammar.

Example:

```
if (condition1 and condition2) {
    int x = 5;
} else {
    int x = 10;
}
```

Grammar:

```
stmt:
  | IF LPAREN expr RPAREN stmt ELSE stmt   { If ($3, $5, $7) }
  | IF LPAREN expr RPAREN stmt         { If ($3, $5, Expr(Noexpr)) }
```

## 5.5 While loops

While loops are identical to C. A while loop is used to repeat a statement or list of statements until a condition evaluates to false.

Example:

```
{| FizzBuzz in Josh. |}
int x = 0;
while (x < 100) {
    string s = “”;
    if (x % 5 == 0) {
        s = s + “Fizz”;
    }
if (x % 3 == 0) {
        s = s + “Buzz”;
    }
    if (s.length == 0) {
        echo(int_to_string(x));
    } else {
        echo(s);
    }
    x = x + 1;
}
```

Grammar:

```
stmt:
  WHILE LPAREN expr RPAREN stmt  { While ($3,$5) }
```

## 5.6 For loops

For loops work similarly to Python’s for-in loops. A for statement loops over the elements of a list and, for each element, executes a statement or list of statements in the body of the loop with the element in scope.

Example:

```
for (x in [1, 2, 3]) {
    int y = x;
    x = x * 2;
    echo(int_to_string(x + y));
}
```

Grammar:

```
stmt:
```

```
  FOR LPAREN ID IN expr RPAREN stmt { For ($3, $5, $7) }
```

## 5.7 Break

Break works like in C and Python. If inside a loop, the `break` keyword exits the most immediate loop even if the iteration of a for loop has not finished or the condition of a while loop is still true.

Example:

```
for (x in myList) {
    if (x == y) {
        break;
    }
    y = y + x;
}
```

Grammar:

```
stmt:
  | BREAK SEMI    { Break }
```

## 5.8 Continue

Continue is like `break`, except instead of exiting the loop completely, it simply goes back to the top of the loop without executing the rest of the statements in the body of the loop. In a for loop, this does mean that it proceeds to the next element.

Example:

```
while (p) {
    if (q) {
        continue;
    }
    total = total + 1;
}
```

Grammar:

```
stmt:
```

```
  | CONTINUE SEMI { Continue }
```

## 5.9 Records

The definition of a record is a statement. Recall the example from the section on non-primitive types.

Example:

```
record Person = {
    string name,
    int age
}
```

Grammar:

```
stmt:
  RECORD ID LBRACE opts_list RBRACE SEMI { RecordDef($2, $4) }

/* for record field and function argument lists */
opts:
  typ ID  { [Opt($1,$2)] }
  | opts COMMA typ ID { Opt($3,$4) :: $1 }

opts_list:
  /* nothing */ { [] }
  | opts { List.rev $1 }
```

## 5.10 Return statements

Functions may return the value of a single expression whose type is consistent with the return type of the function. For a function returning `void`, it is acceptable to write `return;` with no expression. If a return statement is executed, the function exits and is popped from the call stack.

Example:

```
int add(int a, int b) {
    return a + b;
}
```

```
void print(s) {
    echo(s);
    return;
}
```

Grammar:

```
stmt:
  | RETURN SEMI { Return Noexpr }
  | RETURN expr SEMI { Return $2 }
```

# 6 Expressions

## 6.1 Literals

Literal values of primitive types are expressions. See section on Literals.

Example:
`false`, `5`, `2.5`, `'a'`, `"hello"`

Grammar:

```
expr:
   BOOLLIT     { BoolLit $1  }
  | INTLIT     { IntLit $1   }
  | FLOATLIT   { FloatLit $1 }
  | CHARLIT    { CharLit $1  }
  | STRLIT     { StrLit $1   }
  | TRUE       { BoolLit(true)  }
  | FALSE      { BoolLit(false) }
```

## 6.2 Arithmetic operator expressions

The arithmetic operators are used as binary operators of two sub-expressions. They generally return an integer or float depending on context.

Example:
`5 + a`

Grammar:

```
expr:
  | expr PLUS expr      { Binop ($1, Add, $3)   }
  | expr MINUS expr     { Binop ($1, Sub, $3)   }
  | expr DIV expr       { Binop ($1, Div, $3)   }
  | expr MULT expr      { Binop ($1, Mul, $3)   }
  | expr MOD expr       { Binop ($1, Mod, $3)  }
```

## 6.3 Comparisons

The comparison operators are used as binary operators of two sub-expressions. They always return a Boolean.

Example:
`5 > 6, x == y`

Grammar:

```
expr:
  | 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)  }
```

## 6.4 Logical expressions

The logical operators are used as binary operators of two Boolean sub-expressions. They always return a Boolean.

Example:
`true and false`

Grammar:

```
expr:
  | expr AND expr       { Binop ($1, And, $3)   }
  | expr OR expr        { Binop ($1, Or, $3)    }
  | NOT expr        { Unop (Not, $2)    }
```

## 6.5 Assignment

The assignment of a variable (given by its identifier) to the value of an expression is itself an expression. These associate to the right.

Assignment of an expression to a record field or list element is parsed differently (see below).

Example:

```
x = 5;
a = b = c = 10;   {| same as `a = (b = (c = 10))` |}
```

Grammar:

```
expr:
  ID ASSIGN expr   { Assign ($1, $3) }
```

## 6.6 List expressions

### 6.6.1 Creation

### 6.6.2 Access

### 6.6.3 Mutation

## 6.7 Record expressions

### 6.7.1 Creation

### 6.7.2 Access

### 6.7.3 Mutation

## 6.8 Function calls

# 7 Declarations

## 7.1 Variable declarations

Variables must be declared. Optionally, they can be initialized at the same time they are declared. Thus there are two ways to declare a variable:

Example:

```
int x;
int y = 5;
```

Grammar:

```
vdecl:
    typ ID { Declare($1, $2) }
  | typ ID ASSIGN expr { Initialize($1, $2, $4)}
```

### 7.2 Function declarations

Function declarations are identical to those in C.

Example:

```
int main(string args) {
    echo("Hello, World!");
    return 0;
}
```

Grammar:

```
fdecl:
   typ ID LPAREN opts_list RPAREN LBRACE stmt_list RBRACE { { id=$2; params=$4; body
```

## 8 Memory

## 9 Standard Library

Josh's functionality is extended through standard library functions. Users are able to run bash scripts using the `bash` function, which takes as an input the desired script as a string. The output of the command will be returned as a string literal. The command itself will not be parsed. If the user wants to use quotes as part of the command, they have to escape the char with \

Example:

```
String ret = bash(“man -f ls > out.txt | grep -n legacy”)
```

Should the execution of the script fail, the Josh program will continue to run.

We will also implement common commands of UNIX such as echo, cat, grep, cd, ls, rm, and open. These can be called like regular functions in josh.

Example:

```
String ret = cat(“out.txt”)
```

Within the standard library, we also implement functions for the comparison of strings and lists.

Example:

```
list int myList = [1, 2, 3];
list float myList2 = [1, 2, 3, 4];
if (equals(myList, myList2)) (echo(“They are equal!”));
```

## 9.1 Conversions

## 9.2 String operations

## 9.3 List operations

Example:

```
concat([1, 2, 3], [4, 5, 6]);   {| result: [1, 2, 3, 4, 5, 6] |}
```

## 9.4 Bash

## 9.5 I/O

# 10 Appendix

## 10.1 Scanner

```
{ open Parser
  open Scanf }
```

```
let letter = ['a'-'z' 'A'-'Z']
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let escape_char = ''' (escape) '''
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let digit = ['0'-'9']
let id = (letter | '_') (letter | digit | '_')* (* keywords??? *)
let string = '"' ( (ascii | escape)* as lem ) '"'
let char = ''' ( ascii ) ''' (* digit? *)
let float = ('-')? ((digit+) ['.'] digit*) | ((digit*) ['.'] digit+)
let int = ('-')? digit+
let whitespace = [' ' '\t' '\r' '\n']

rule token = parse
  | whitespace { token lexbuf }
  | "{|"     { comment lexbuf }
  | '('      { LPAREN }
  | ')'      { RPAREN }
  | '{'      { LBRACE }
  | '}'      { RBRACE }
  | '['      { LBRACK }
  | ']'      { RBRACK }
  | '.'      { DOT }
  | ';'      { SEMI }
  | ','      { COMMA }
  | '+'      { PLUS }
  | '-'      { MINUS }
  | '*'      { MULT }
  | '/'      { DIV }
  | '%'      { MOD }
  | '='      { ASSIGN }
  | "=="     { EQ }
  | "!="     { NEQ }
  | '<'      { LT }
  | "<="     { LEQ }
  | '>'      { GT }
  | ">="     { GEQ }
  | "and"    { AND }
  | "or"     { OR }
```

```
  | "not"    { NOT }
  | "if"     { IF }
  | "else"   { ELSE }
  | "for"    { FOR }
  | "in"     { IN }
  | "while"  { WHILE }
  | "break"  { BREAK }
  | "continue" { CONTINUE }
  | "return" { RETURN }
  | "int"    { INT }
  | "bool"   { BOOL }
  | "float"  { FLOAT }
  | "string" { STRING }
  | "void"   { VOID }
  | "char"   { CHAR }
  | "true"   { TRUE }
  | "false"  { FALSE }
  | "record" { RECORD }
  | int as lem { INTLIT(int_of_string lem) }
  | float as lem { FLOATLIT(float_of_string lem)}
  | char as lem { CHARLIT(String.get lem 1)}
  | string { STRLIT(Scanf.unescaped lem)}
 | escape_char as lem { CHARLIT(String.get (Scanf.unescaped lem) 1) }
  | id as lem { ID(lem) }
  | eof { EOF }
 | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

## 10.2 Parser

```
/* Ocamlyacc parser for Josh */

%{
open Ast
```

```
%}

%token SEMI LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE COMMA DOT
%token PLUS MINUS ASSIGN MULT DIV MOD
%token EQ NEQ LT GT LEQ GEQ AND OR NOT
%token IF ELSE FOR IN WHILE BREAK CONTINUE RETURN
%token INT BOOL FLOAT STRING VOID CHAR
%token RECORD
%token <int> INTLIT
%token <char> CHARLIT
%token <string> STRLIT
%token <bool> BOOLLIT
%token TRUE FALSE
%token <float> FLOATLIT
%token <string> ID
%token EOF

%start program
%type <Ast.program> program

%right ASSIGN
%left NOT
%left OR
%left AND
%left EQ NEQ
%left LT LEQ
%left GT GEQ
%left MOD
%left MULT DIV
%left PLUS MINUS

%%

program:
  top_level_list EOF { $1 }

top_level_list:
  { [] }
```

```
  | top_level top_level_list { $1::$2 }

top_level:
    stmt { Stmt $1 }
  | fdecl { Fdecl $1 }

typ:
  INT       { Int  }
  | BOOL    { Bool }
  | FLOAT   { Float }
  | CHAR    { Char }
  | STRING  { String }
  | LBRACK typ RBRACK { ListT($2) }
  | VOID    { Void }
  | ID { RecordType($1) }
  | typ ID LPAREN opts_list RPAREN { FunkType($2, $4, $1) }
      /*
      record Thing {
          string name;
          int fptr(char c, float d);
      }
      record Thing myThing = { "josh", func };
      myThing.func(c, d);

      */

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

fdecl:
   typ ID LPAREN opts_list RPAREN LBRACE stmt_list RBRACE { { id=$2; params=$4; body

stmt:
  expr SEMI                                  { Expr $1       }
  | vdecl SEMI                                    { Vdecl $1 }
  | LBRACE stmt_list RBRACE                   { Block $2      }
  | IF LPAREN expr RPAREN stmt ELSE stmt   { If ($3, $5, $7) }
```

```
  | IF LPAREN expr RPAREN stmt         { If ($3, $5, Expr(Noexpr)) }
   | FOR LPAREN ID IN expr RPAREN stmt { For ($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt            { While ($3,$5)   }
  | RECORD ID LBRACE opts_list RBRACE SEMI { RecordDef($2, $4) }
   | RETURN SEMI { Return Noexpr }
   | RETURN expr SEMI { Return $2 }
   | CONTINUE SEMI { Continue }
   | BREAK SEMI    { Break }

expr_list:
    /* nothing */   { [] }
    | expr_list COMMA expr { $3::$1 }

expr:
  /* literals */
   BOOLLIT                      { BoolLit $1  }
  | INTLIT                       { IntLit $1   }
  | FLOATLIT                     { FloatLit $1 }
  | CHARLIT                      { CharLit $1  }
  | STRLIT                       { StrLit $1   }
  | TRUE             { BoolLit(true)  }
  | FALSE            { BoolLit(false) }
  | ID                            { Id $1         }
  /* arithmetic expressions */
  | expr PLUS expr      { Binop ($1, Add, $3)   }
  | expr MINUS expr     { Binop ($1, Sub, $3)   }
  | expr DIV expr       { Binop ($1, Div, $3)   }
  | expr MULT expr      { Binop ($1, Mul, $3)   }
  | expr MOD expr        { Binop ($1, Mod, $3)  }
  /* equality */
  | 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)  }
  /* logical */
  | expr AND expr       { Binop ($1, And, $3)   }
```

```
  | expr OR expr        { Binop ($1, Or, $3)    }
  | NOT expr        { Unop (Not, $2)    }
  | ID ASSIGN expr           { Assign ($1, $3)       }
  | LPAREN expr RPAREN       { $2                    }
  /* list */
  | LBRACK expr_list RBRACK { ListLit(List.rev $2)  }
  | expr LBRACK expr RBRACK { ListAccess($1, $3) }
  | expr DOT ID { RecordAccess($1, $3) }
  /* record instantiation */
  | ID LBRACE actuals_list RBRACE { RecordCreate($1, $3) }
  /* mutation */
  | expr DOT ID ASSIGN expr { MutateRecord(($1,$3), $5) }
 | expr LBRACK expr RBRACK ASSIGN expr { MutateList(($1,$3), $6) }
  /* function call */
  | ID LPAREN actuals_list RPAREN { Call($1, $3) }
 | expr DOT ID LPAREN actuals_list RPAREN { CallRecord(($1,$3), $5) }
 | expr LBRACK expr RBRACK LPAREN actuals_list RPAREN { CallList(($1,$3), $6) }

vdecl:
    typ ID { Declare($1, $2) }
  | typ ID ASSIGN expr { Initialize($1, $2, $4)}

/* for record field and function argument lists */
opts:
    typ ID  { [Opt($1,$2)] }
    | opts COMMA typ ID { Opt($3,$4) :: $1 }

opts_list:
    /* nothing */ { [] }
  | opts { List.rev $1 }

/* for instantiating records and calling functions */
actuals_list:
    /* nothing */ { [] }
    | actuals_list COMMA actual { $3::$1 }

actual:
    expr { Actual $1 }
```

### 10.3 AST (work in progress)

# 11 References

Rusty LRM C LRM