

Extend

Language Reference Manual

Ishaan Kolluri, Kevin Ye, Jared Samet, Nigel Schuster

October 21, 2016

Contents

1	Introduction to Extend	2
2	Structure of an Extend Program	2
3	Types and Literals	8
3.1	Primitive Data Types	8
3.2	Ranges	9
3.2.1	Range Slicing	9
3.2.2	The Underscore Symbol	10
4	Expressions	11
4.1	Operators	11
4.1.1	Multiplication	11
4.1.2	Addition	11
4.1.3	Unary	11
4.2	Booleans	11
4.3	Variable Declaration	11
4.4	Variable Assignment	11
4.5	Conditionals	11
5	Functions	11
5.1	Format	11
5.2	Dimension Assignment	12
5.3	Application on Ranges	13
5.4	Dependencies Illustrated	14
6	File I/O	15
6.1	File	15
6.2	Input Arguments	15

6.2.1	How to run a program	15
6.2.2	Main function	15
7	Example Program	15

1. Introduction to Extend

Extend is a domain-specific programming language used to designate ranges of cells as reusable functions. It abstracts dependencies between cells and models a dependency graph during compilation. In order to offer great performance for any size of datasets, Extend compiles down to LLVM.

Extend’s syntax is meant to provide clear punctuation and easily understandable cell range access specifications, while maintaining the look of modern functional programming languages. Given Extend’s functionality resonates well with spreadsheets, it borrows syntactical elements from programs such as Microsoft Excel.

2. Structure of an Extend Program

Extend is predominantly composed of function declarations. In order to run the program, the **main** function will be executed. To illustrate the scope of the language, the OCaml grammar is attached below:

```
/* Ocamlyacc parser for Extend */

%{
open Ast
%}

%token LSQBCK RSQBCK LPAREN RPAREN LBRACE RBRACE
%token COLON COMMA QUESTION GETS EQ SEMI UNDERSCORE
```

%token SWITCH CASE DEFAULT

%token PLUS MINUS TIMES DIVIDE MOD

%token EMPTY RETURN

%token <int> LIT_INT

%token <float> LIT_FLOAT

%token <string> ID

%token EOF

%right QUESTION

%left PLUS MINUS

%left TIMES DIVIDE MOD

%start program

%type <Ast.program> program

%%

program :

func_decls EOF { List.rev \$1 }

func_decls :

/* nothing */ { [] }

| func_decls func_decl { \$2 :: \$1 }

func_decl :

ID LPAREN func_param_list RPAREN LBRACE opt_stmt_list ret_stmt RBRACE

{ ((1,1), \$1, \$3, \$6, \$7) }

| ret_dim ID LPAREN func_param_list RPAREN LBRACE opt_stmt_list ret_stmt RBRACE

{ (\$1, \$2, \$4, \$7, \$8) }

opt_stmt_list:

/* nothing */ { [] }
| stmt_list { List.rev \$1 }

stmt_list:

stmt { [\$1] }
| stmt_list stmt { \$2 :: \$1 }

stmt:

vardecl { \$1 } | assign { \$1 }

ret_stmt:

RETURN expr SEMI {\$2}

vardecl:

ID varassign SEMI {(\$1, \$2)}
| dim ID varassign SEMI {(\$1, \$2, \$3)}

varassign:

/* nothing */ {}
| GETS expr {(\$2)}

assign:

ID lhs_sel EQ expr SEMI { (\$1,\$2,\$4) }

expr:

```

ID rhs_sel {($1,$2)}
| op_expr { $1 }
| ternary_expr { $1 }
| switch_expr { $1 }
| func_expr { $1 }
| LPAREN expr RPAREN { $2 }
| LIT_INT { $1 }
| LIT_FLOAT { $1 }
| EMPTY { Empty }

```

op_expr :

```

expr PLUS expr { ($1, $3) }
| expr MINUS expr { ($1, $3) }
| expr TIMES expr { ($1, $3) }
| expr DIVIDE expr { ($1, $3) }
| expr MOD expr { ($1, $3) }

```

ternary_expr :

```

/* commented out optional part for now */
expr QUESTION expr COLON expr %prec QUESTION { ($1, $3, $5) }

```

switch_expr :

```

SWITCH switch_cond LBRACE case_list RBRACE { ($2, List.rev $4) }

```

switch_cond :

```

/* nothing */ { True }
| expr { $1 }

```

case_list:

```
    case_stmt { [$1] }  
| case_list case_stmt { $2 :: $1 }
```

case_stmt:

```
    DEFAULT COLON expr SEMI { $3 }  
| CASE case_expr_list COLON expr SEMI { (List.rev $2, $4) }
```

case_expr_list:

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

func_expr:

```
    ID LPAREN opt_arg_list RPAREN { $3 }
```

opt_arg_list:

```
    /* nothing */ { [] }  
| arg_list { List.rev $1 }
```

arg_list:

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

lhs_sel:

```
    /* nothing */ { [0,0] }  
| LSQBRACT lslice COMMA lslice RSQBRACT { ($2,$4) }  
| LSQBRACT lslice RSQBRACT { ($2) }
```

rhs_sel:

```
/* nothing */ { [0,0] }  
| LSQBRACK rslice COMMA rslice RSQBRACK { ($2,$4) }  
| LSQBRACK rslice RSQBRACK { ($2) }
```

lslice:

```
/* nothing */ { 0 }  
| lslice_val { $1 }  
| lslice_val COLON lslice_val { ($1,$3) }
```

rslice:

```
/* nothing */ { 0 }  
| rslice_val { $1 }  
| rslice_val COLON rslice_val { ($1,$3) }
```

lslice_val:

```
expr { $1 }
```

rslice_val:

```
expr { $1 }  
| LSQBRACK expr RSQBRACK { ($2) }
```

func_param_list:

```
/* nothing */ { [] }  
| func_param_int_list { List.rev $1 }
```

func_param_int_list:

```
func_sin_param { [$1] }
```



```
| func_param_int_list COMMA func_sin_param { $3 :: $1 }
```

func_sin_param :

```
ID { ($1) }
```

```
| dim ID { ($1, $2) }
```

dim :

```
LSQBRACK lslice_val RSQBRACK { $2 }
```

```
| LSQBRACK lslice_val COMMA lslice_val RSQBRACK { ($2,$4) }
```

ret_dim :

```
LSQBRACK ret_sin COMMA ret_sin RSQBRACK { ($2,$4) }
```

ret_sin :

```
LIT_INT { $1 }
```

```
| ID { $1 }
```

```
| UNDERSCORE { }
```

3. Types and Literals

3.1. Primitive Data Types

Extend's basic primitives are *integers*, *floats*, *char* literals, and *string* literals. They are all internally represented as numbers or a range of numbers. They are as follows:

Char

A **char** literal is essentially a size 1 numerical range. At evaluation, the number in the range will be compared with its ASCII equivalent.

String

A **string** literal is a range of numbers of size n , where n is the length of the string. The string

'hello' can be represented internally as [104, 101, 108, 108, 111].

Integer

A **integer** can be represented as a size 1 numerical range as well. However, it retains its numerical value upon evaluation.

Float

A **float**, like Javascript numbers, can be represented as 64 bit, where the fraction is stored in bits 52 to 62.

Below is a snippet illustrating programmatic declarations for each of the above types.

```
/* Integer */
num = 5;

/* Char */
chr = 'A'

/* String */
str = 'Hello'

/* Float */
num = 1.5;
```

3.2. Ranges

Ranges are a data type unique to the Extend language. It borrows conceptually from spreadsheets; a range is a group of cells with dimensions represented as rows and columns. Each range is either one or two-dimensional. A range is composed of cells, and cells are comprised of functions that can have dependencies on the values of other cells. A range is written as follows:

```
/* This is a left-handed range, used to assign a value. */
[1,2]foo; /*Range with 1 row and 2 columns */
```

3.2.1. Range Slicing

Extend somewhat mimics Python in its range slicing syntax; however, it offers the ability to slice a range in both absolute and relative terms.

```

foo[1,2] /* This evaluates to the cell value at row 1, column 2. */
foo[1,] /* Evaluates to the range of cells in row 1. */
foo[,2] /* Evaluates to the range of cells in column 2.*/
foo[, [1]] /* The internal brackets denote RELATIVE notation.
In this case, 1 column right of the one currently being operated on. */
foo[5:, 7:] /* 5th row down, and 7th column from the absolute origin.
foo[[1:2], [5:7]]
/* Selects the rows between the 1st and 2nd row from current row */
/* Selects the columns between 5th and 7th column from current column */

```

3.2.2. The Underscore Symbol

The underscore(_) symbol allows the dimension of the range to have an unspecified size. For example, in function signatures, using the underscore allows the return value to have various possible dimensions. An example is illustrated below:

```

[1, _]foo;
/* A range with 1 row and an unknown, variable number of columns. */

```

4. Expressions

4.1. Operators

4.1.1. Multiplication

4.1.2. Addition

4.1.3. Unary

4.2. Booleans

4.3. Variable Declaration

4.4. Variable Assignment

4.5. Conditionals

5. Functions

Functions lie at Extend's core; however, they are not *first class objects*. Since it can be verbose to write certain operations in Extend, the language will feature a comprehensive number of built-in and standard library function. An important built-in function will be I/O (see section 6).

5.1. Format

Every function in Extend follows the same format, but allows some optional declarations. As in most programming languages the header of the function declares the parameters it accepts and the return type. The simplest function is this:

```
[1,1] foo([1,1] arg) {  
    return arg;  
}
```

This function simply returns whatever value is passed into it. The leading `[1,1]` marks the return dimensions. `foo` is the function name. In parentheses the function arguments are declared, again

with dimensions of the input. The body of the function follows, which in this case is only the return statement.

5.2. Dimension Assignment

Extend will feature gradual typing for function declarations. This will enable users with a weak experience in typing to use the language, but maintains the ability to improve during development.

To avoid specifying the return dimensions, an underscore can be used. This marks a variable range. Thus our function now looks like this:

```
[_, 1] foo ([5, 5] arg1, [1, 1] arg2) {  
    return arg1[0:arg2_, 0];  
}
```

Here we are selecting a range from `arg1` that depends on the value of `arg2` and can therefore not be known ahead of time.

However Extend will feature even more options to specify ranges. If a certain operation should be applied to a range of numbers of unknown size, the size can be inferred at runtime and match the return size:

```
[m, 1] foo ([m, 1] arg) {  
    return arg[0:m-1_, 0] + 1;  
}
```

This function will add 1 to each element in `arg`. Notice, that `m` is used across the function to apply the operation to the range.

Summarizing, we have 3 ways of specifying a return range:

Type	Symbol Example	Description
Number	3	A number is the simplest descriptor. It specifies the absolute return size
Variable	bar	A variable identifier. To use this, the identifier must also be present as a range descriptor in a function parameter.
Underscore	-	This marker is unique, since it is a wildcard. While the other options aim to be specific, the underscore circumvents declaring the range size.

5.3. Application on Ranges

Extend gives the developer the power to easily apply operations in a functional style on ranges. As outlined in the section above, there are various ways to apply functions to ranges. A feature unique to Extend is the powerful operation on values and ranges. To apply a function on a per cell basis, the corresponding variable needs to be preceded by "#". The following function applies cell wise addition:

```
[m,n] foo ([m,n] arg1 , [m,n] arg2) {
    [m,n] bar = #arg1 + #arg2;
    return bar;
}
```

If we want to apply a function to the whole range at once we drop the leading symbol. Thus matrix addition takes the following shape:

```
[m,n] foo ([m,n] arg1 , [m,n] arg2) {
    [m,n] bar = arg1 + arg2;
    return bar;
}
```

While both function above result in the same value, and only show the syntactical difference, more involved functions. If we wanted each cell to to be the square root divided by the sum of the input

we have the following:

```
[m] foo ([m] arg) {  
    [m] bar = sqrt(#arg) / sum(arg);  
    return bar;  
}
```

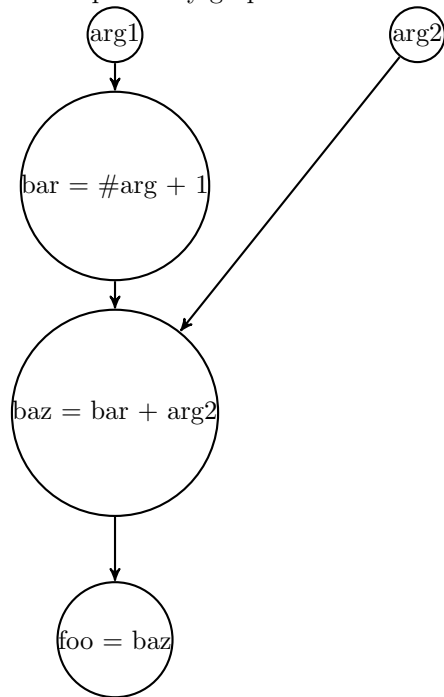
Notice that `arg` is only once preceded by `#`.

5.4. Dependencies Illustrated

The dependency resolution is another asset that sets Extend apart from other languages. Most languages compile ordinarily and execute the given commands sequentially. Extend builds a dependency graph. The advantage of this is that only relevant code segments will be executed. Given the function

```
[m,n] foo ([m,n] arg1, [m,n] arg2) {  
    [m,n] bar := #arg1 + 1;  
    [m,n] faz := #arg1 + 3;  
    [m,n] baz := bar + arg2;  
    return baz;  
}
```

The dependency graph will look like this:



Notice that `faz` does not appear in the graph, because it is not relevant for the return value.

Ultimately this graph enables the Extend to find the leaves, evaluate code paths in the best configuration and even in parallel.

6. File I/O

6.1. File

6.2. Input Arguments

6.2.1. How to run a program

6.2.2. Main function

7. Example Program