# FASTER

Programming Language and translators 4115

Anand Rajan (asr2171)

August 22nd 2014

## Table of Contents

## 1. Introduction

FASTER is a programming language that focuses on making parallel programs based on collections easier to write. Given a collection (of either ints or structs), a user can write an easier syntax to parallelize this array/collection. The language will have some similarity to C and the similarities and differences will be described in the following sections.

## 2. Language Tutorial

Here is a quick example of an example FASTER program

```
main()
{
    int i;
    i = 0;
    print("my first FASTER program");
    print(i);
}
```

As you can see it looks similar enough to C, and now spicing things up a bit further to show some different features of the language.

```
main()
{
    int i;
    int b;
    struct test2 arraytest[40];
    struct test2 arraytest1[40];

    struct test2 x;
    int a[40];
    arraytest[1].a = 1;

    for (i =0; i < 40; i = i +1)
    {
        arraytest[i].a = i;
        arraytest[i].b = i ;
    }


    arraytest:
    {a:
        print(a.b);
        print(" ");
    };
    arraytest1:
    {a:
        a.b = 0;
        print("no");
    };



}




struct test2
{
    int a;
    int b;
};
```

As one can see in heart of FASTER is this parallel notation for arrays

```
    arraytest:
        {a:
```

| |
|---|
| print(a.b); |
| print(" "); |
| }; |

As one can see this notation is neat and concise and takes the strain away from using the loop from the user.

## 2.1 How to Compile and Run FASTER

First FASTER programs use the file extension .fs. The ocaml executable that is generated will be named faster. To interact with FASTER you have multiple options.

| Argument | Description |
|---|---|
| -b < file_name | Generates the byte code of the program |
| < file_name | This prints the C code to standard output and also outputs the C code to a faster_generated.c file |
| Make rtest param=filename | Also to make and execute the C code of the program this option is provided, this will generate the C code and then execute the C code. |

One can also run gcc –fopenmp faster_generated.c to compile the generated.c file themselves to see if there are any issues.

# 3. Language reference Manual

## 3.1 Lexical Conventions

There are six kinds of tokens: identifiers, keywords, constants, expression operators and other separators. In general blanks, tabs, newlines and comments as described below are ignored except they serve as token separators. At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.

### 3.0.1 Comments

The characters /* introduce a comment, which terminates with the characters */. FASTER does not support comment nesting.

### 3.0.2 Identifiers

An identifier is a sequence of letters and digits, but the first character must be alphabetic. The underscore "_" character counts as an alphabetic character. Upper and lower case characters are different, such that foo and FOO are considered different identifiers.

### 3.0.3 Constants

#### 3.0.3.1 *Integer Constants*

An integer constant is a sequence of digits. An example would be 23. FASTER does not allow negative integers.

#### 3.0.3.2 *String constants*

A string is a sequence of zero or more characters surrounded by double quotes " ". String constants in FASTER can only be used as a parameter in the print method. They have no other role to play in terms of declaration, assignment or manipulation of a string.

### 3.0.4 Primitive types

The only primitive types in FASTER that can be used is an int and manipulated with an int. As mentioned previously string constants are only used to display strings when debugging. The way you declare your ints are with int following the name of the indentifier, after which there would be a semi colon.
So for example declaring and assigning to this identifier x would be

|  |
|---|
| int x; |
| x =1; |
|  |

### 3.0.5 Structures

A structure is a programmer-defined data type made up of other primitive types (only ints in this cases as the language only supports ints. Having other arrays or structures inside structures are not allowed in FASTER

### *3.0.6 Defining Structures*

You can define structure using the struct keyword followed by the declarations of structures members enclosed in braces after which a semicolon is necessary. You declare each member of structure just like you normally declare a variable with, data-type variable name followed with a semi colon. You should also include a name for your structure in between the struct keyword and the opening brace. In this following case a struct stock is declared with two primitive types defined in it, price and quantity.

```
struct stock
{
int price;
int quantity;
};
```

### *3.0.7 Declaring Struct variables*

To declare struct variables in your code, you use the struct keyword followed by the struct name you had defined the struct (in the above example), followed an identifier variable that you can come up with. So an example of this would be below.

```
int main()
{
 struct stock s1;
 s1.price = 20;
}
```

In the example above a variable with variable name s1 is defined, which is of type (struct stock). The member expression to get to price member is shown above, this will be discussed in greater detail in the Member expressions part (See 3.4.1).

### 3.0.8 Arrays

An array is a data structure that lets you store one or more elements consecutively in memory. In FASTER, the array elements are indexed beginning at position zero, not one.

### 3.0.9 Declaring Arrays

You declare an array specifying the data type for its elements, its name and the number of elements it can store in it. In FASTER the data types that arrays work on are the primitive types and also any structs that you define.

```
int main()
{
      int test_array[10];
      struct stock stocks[100];
}
```

In the above example you declare an "int array" that contains 10 elements. And then continuing the stock example we had from the beginning, we declare an array of stock structures here.

### *3.0.10 Accessing Array elements*

You can access the elements of the array by specifying the array name followed by the element index in square brackets. Remember that the array elements are numbered starting with zero. So an example of this is below

```
int main()
{
      struct stock stocks[100];
      stocks[0].price = 100;
}
```

In the above stock array defined, the first element of the stock is accessed which is of type stock, and then its price integer is set to 100.

## 3.1 Objects and Lvalues

An object is a region of storage that can be manipulated; a lvalue is an expression referring to an object. An obvious example of an lvalue expression is an identifier. The name lvalue comes from the assignment expression "E1 = E2" in which the left operand E1 must be an lvalue expression. These terms will be brought up again in the following sections.

## 3.2 Expressions and Operators

An expression consists of atleast operand and zero or more operators. Some examples are 3 or 2+2. In this section the different type of expressions and operators will be noted. One thing to note is that A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

### 3.2.1 Member Access Expressions

You can use the member access operator . to access the members of a structure. You put the name of the structure variable on the left side of the operator, and the name of the member in the right side of the operator.

```
int main()
{
 struct stock s1;
 s1.price = 20;
}
```

In the following example (continuing on our stock example), the price of s1 is set to 20 via the member access expression.

### 3.2.2 Array Subscripts

You can access array elements by specifying the name of the array, and the array subscript enclosed in brackets. Here is an example below that shows how the first element (zero'th index) of the array stocks is accessed and used.

```
int main()
{
	struct stock stocks[100];
	stocks[0].price = 100;
}
```

### 3.2.3 Multiplicative, Divide and Mod Operators

The multiplicative operators are * and / and the % mode operators they group from left to right.

#### *3.2.3.1 Expression * Expression*

The binary operator * indicates multiplication. If both operands are of type in, then this result int is allowed. No other combinations are allowed

#### *3.2.3.2 Expression / Expression*

The binary operator / indicates division. The same considerations as multiplication apply here.

### 3.2.4 Additive Operators

The additive operators + and – group from the left to right.

3.2.4.1 Expression + Expression
The result will be the sum of these expressions. So the same type check rules that applied to multiplication apply here too.

3.2.4.2 Expression – Expression
Here the binary operator – indicates subtraction. The same checking rules as multiplication apply here.

### 3.2.5 Relational Operators

The relational operators < (Less than), > (Greater than), <= (Less than equal to), >= (Greater than equal to) group from left to right. More interestingly here, they only apply to ints again. And the result of these relations will result in 0 or 1.

### 3.2.6 Equality Operators

The equality operators == (equal to) and != (not equal to) group from left to right and again follow the rules of the multiplicative operators, in that only 'int' types are allowed. This may seem overly restrictive but I wanted them to follow the same pattern as the rest of the binary operators. Also without the notion of pointers in the language, the notion of equality may get muddled, so I went with the route of this restriction.

### 3.2.7 Assignment Operators

The assignment operator = groups from right to left (right associative). All this requires for it to succeed taking the below as an example
Lvalue = expression
It requires an lvalue as its left operand, and the lvalue's type should match the type of the assigned expression. The value of the expression replaces that of the object referred to by the lvalue. So all types in the system can get assigned to one another including ints, arrays and structs.

### 3.2.8 Operator Precedence

The following table describes the operator precedence that for the operators that were explained in detail above

| Precedence | Expressions / Operators |
|---|---|
| Highest (left associative) | F (arg,arg…) Dot operator (.) Array subscripts ([]) and also parenthesis () |
| (left associative) | * (Multiply) / (Divide) % (Mod) |
| (left associative) | + - (Add subtract) |
| (left associative) | Less than, Greater than, Less than equal to, Greater than equal to ( <, > <=,>=) |
| (left associative) | == != ( l ==r , l != r) |
| Lowest (right associative) | Assign ( l = r) |
| | |

## 3.3 Statements

### 3.3.1 Expression statements

You can turn any expression into a statement by adding a semicolon to the end of the expression. Here are some examples:

```
5;
2 + 2;
10 >= 9;
```

Expression statements are only useful when they have some kind of side effect, such as storing a value, calling a function, or (this is esoteric) causing a fault in the program. Here are some more useful examples:

```
int y;
int x;
y = x + 25;
```

### 3.3.2 The If Statement

You can use the if statement to conditionally execute part of your program, based on the truth value of the expression. Here is the general form of the if statement.

```
if (test)
then-statement
else
```

```
else-statement
```

If test evaluates to true, then then-statement is executed and else-statement is not. If test evaluates to false, then else-statement is executed and then-statement is not. The else clause is optional.

### 3.3.3 For Statement

The for statement is a loop statement whose structure allows easy variable initialization, expression testing, and variable modification. Its very convenient for making counter controlled loops. Here is the general form of the for statement.

```
for (initialize; test; step)
statement
```

The for statement first evaluates the expression initialize. Then it evaluates the expression test. If test is false, then the loop ends and the program resumes after statement. Otherwise if test is true, the statement is executed. Finally step is evaluated, and the next iteration of the loop begins evaluating the test again.

More often than not initialize step assigns values to a variable that are then generally used as counters.

### 3.3.4 Parallel Array Access:

The generalized form of the parallel access syntax of the array is as follows:

```
Array:{
Id:
        Statements modifying Id;
}
```

So the syntax has an array variable followed by a colon, followed by a left curly brace, and then followed that an identifier, that indicates each element of the array. Then there is another colon and then there are a bunch of statements that modify the Id potentially and ending again with a close curly brace.

This program is a good example that encapsulates most of the constructs we have talked about in this language reference manual.

```
struct stock
{
    int price;
    int quantity;
    int calc;
};
```

```
main()
{
        int i;
        struct stock stocks[100];

        for (i =0; i < 100; i = i +1)
        {
                stocks[i].price = i * 3;
                stocks[i].quantity = i * 7;
        }

        stocks:
        {s:
                s.calc = s.price * s.quantity * 17 * 34 * 56 - 45 +123;
                print (s.calc);
        };
}
```

Here stocks variable is an array of type stock structs is declared initially. The different elements of stocks are then initialized to somewhat different quantities. Then using the parallel array syntax values in the calc element of stocks is calculated using the above expressions (s.price * s.quantity * 17 …. ). This is then printed out using the print function. So without the user straining themselves to think about openmp or worry to include any other libraries, this array syntax is available for them to use.

### 3.3.5 While

The while statement is a loop statement with an exit test at the beginning of the loop. Here is the general form of the while statement:

```
while (test)
statement
```

The while statement first evaluates test. If test evaluates to true, statement is executed, and then test is evaluated again. The statement continues to execute repeatedly as long as test is true after each execution of statement.

## 3.4 Functions

You can write functions to separate parts of your program into distinct subprocedures. To write a function, you must at least create a function definition. Every program requires at least one function, called main. That is where the program’s execution begins. There are couple of built in functions print and exit(). Print allows integers and strings to be printed and prints to the standard output whereas exit() calls the system call exit and exits from the program.

### 3.4.1 Function Declaration/Definition

You declare and define a function by specifying the name of a function, a list of parameters, and the function's return type. And then begins the body of the function. Here is the general form:

```
return-type function-name (parameter-list) { function-body }
```

return-type indicates the data type of the value returned by the function. You can declare a function that doesn't return anything by using the return type void. Unfortunately in faster as of now only void type is implemented, so your functions don't return anything.

function-name can be any valid identifier

parameter-list consists of zero or more parameters, separated by commas. A typical rameter consists of a data type and an optional name for the parameter

A function-body is simply just a series of statements.

### 3.4.2 Calling functions

You can call a function by using its name and supplying any needed parameters. Here is the general form of a function call:

```
function-name (parameters)
```

## 3.5 Program Structure and Scope

A FASTER program must exist solely in a single source file. The scope of local variables is that they will only be visible in the function they are defined. There will be errors in duplicate definitions of local variables. The only exception to this scope is the scope of the parallel access arrays. Local variables in the function are not available inside the parallel access, array, and the parallel access loop variable is not available to the function outside.

## 3.6 Example

Finally an example is given of the program, which is an example of parallel linear search, which continues on from our stock example. So we have an array of stocks and this is parallel used to find the desired quantity (379) in an easy parallel fashion, without the user needing to even think of any parallelism at all.

```
void main()
{
        int i;
        struct stock stocks[1000];

        for (i =0; i < 1000; i = i +1)
        {
                stocks[i].price = i;
                stocks[i].quantity = i * 17;
        }

        stocks:
        {s:
                if (s.price == 379)
                {
                        print(s.quantity);
                        print("\n");
                        exit();
                }
        };
}

struct stock
{
        int price;
        int quantity;
        int calc;
};
```

Another example is parallel prime factorization as shown below,

```
void main()

{       int b[99987147];

        int i;

        b[0] = 2;
        b[1] = 2;

        for (i = 0; i < 99987147; i = i+1)

        {

              if (i > 1)

              {

                    b[i] = i;

              }



        }



        b:

        {b1:

               if (99987147 % b1 == 0)

              {

                    print (b1);

                    print (" is a prime factor \n");


              }



        };



  }
```

~

~

# 4. Project Plan

## 4.1 Project timeline

| Date | Milestone |
|---|---|
| June 11th | Initial proposal |
| July 2nd | LRM Due |
| July 19th | Beginning to dig deeper and try understand Ocaml |
| July 25th | Understood Microc |
| July 29th | Started to work on parser.mly, scanner.mll and ast.ml |
| Aug 2nd | Ast, parser and scanner ready |
| Aug 2nd | Begin to work on the semantic.ml |
| Aug 9th | Semantic.ml done begin to work on compile.ml and byte code generation. Keep working on test cases in the same time. |
| August 16th | Code generation working realize need to revise some features after Prof's advise. Attempting to revise on them. |
| August 18th | Work on final report |

## 4.2 Software Development Environment

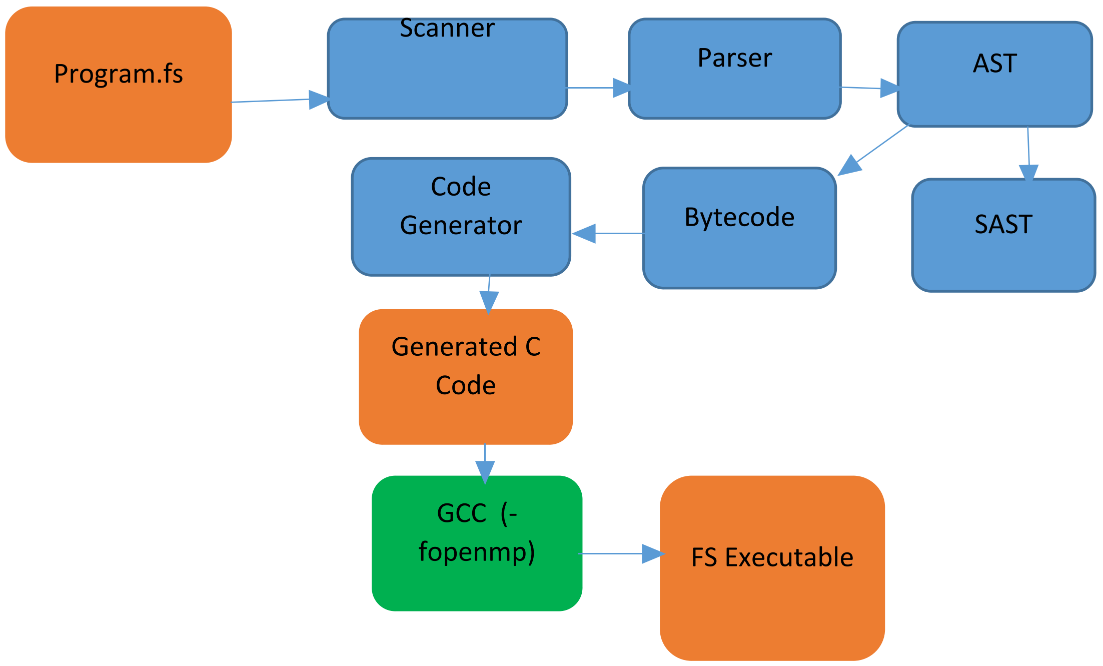
a. Development environment on vim in Linux VM environment.
b. Programming environment Ocaml 3.12.1
c. Gcc Compiler on linux to compile the generated C code
d.

## 4.1 Processes followed

1. Version control: Use P4V and backed on NEC depository at work. This is backed up on tape in different intervals of the day.
2. Coding style for Ocaml:
   a. Use Vim indentation for when the are nested if statements in a function
   b. Try to break down complex functions into smaller functions
   c. Try to keep one statement per line.
   d. Each code block following let in should be indendented.

# 5 Architecture

## 5.1 Overview



The architecture of FS is shown above. The input and output files are marked in orange. Overall FASTER follows a traditional compiler model with a lexical analyzer and parser at the front end, followed by generation of a semantically checked and typed abstract syntax tree (SAST) from an abstract syntax tree (AST) and finally bytecode generation. This byte code generation is then followed by the code generator taking these byte codes and converting them into C code. GCC then with the –fopenmp directive is used to convert the generated C code into an FS executable.

## 5.2 Scanning, parsing and AST

The scanner creates lexical tokens from the stream of input character from the input program file. These tokens are then interpreted by the parser according to the precedence rules of the FASTER language. The parser's main goal is to organize the tokens of the program into 2 record lists: function declarations and structure declarations. Within each record lies the respective declarations along with the name and type information of the data structures. Specific to the function declarations record is the creation of an AST of functions from groups of statements, statements evaluating the results of expressions, and expressions formed from operations and assignments of variables, references and constants. The below code snippet shows the expressions in the faster language.

```
type expr =
   Literal of int
  | Id of string
  | Binop of expr * op * expr
  | Assign of expr * expr
  | Call of string * expr list
  | Array of expr * expr
  | MemberAccess of expr * op * string
  | Noexpr
  | String of string
```

## 5.3 SAST Creation

For each function, the SAST component creates a series of function, structure and local indexes which hold information about the types and names of the expressions on the leaves of the AST. Starting with the leaves of the AST, each function, reference, variable or constant is assigned a type. Expressions using these values are then assigned a type based on the operation performed. As each node in the AST assigned a type, a series of type checks is performed based on the operation being applied. A summary of checks is given below.

| Checks | Error |
|---|---|
| If Conditions | The conditional expression must be int |
| Variable assignments | Left and right hand side must match |
| Variable declarations | Variables can not be declared twice |
| Structs/Functions | Structs can not have the same name as functions, nor can there be repeated function names nor can there be repeated struct names. |
| Binary operations | The left and right hand side has to be integers |
| Function arguments | The function arguments must type match |
| Function return | The return arguments must type match |

## 5.4 Compiler / Byte code generation

The main function of the compiler is to convert the AST tree into a flattened list of bytecode nodes. The advantage of the flattened bytecode list, is that the code generator can use that solely to generate codes.

## 5.5 Generator

The main job of the generator is to generate the C code from the bytecodes, looking at each of the statements of the bytecodes.

# 6. Test Plan

### 6.1 Unit testing

Unit Testing was done in frequent intervals, where each unit being made was unit tested rigorously with multiple cases. The scanner, parser and AST were first tested and then in the later phase the semantic checker, the compiler and the code generator were tested.

## 6.2 Regression testing

While hopping and heaving and changing features, the tests were useful barometer of regression, to see how much impact did the last minute changes have and so were useful in their own perspective.

## 6.3 Automation

A shell script (provided by our dear lecturer) was provided to automate test cases and was use readily in each stage of our design.

Following are the list of test cases, shown below. The ones that are marked as fail we actually expect to fail.

```
fail-test-duplicate-struct-name.fs

fail-test-param1.fs

test-argscrazy.fs

test-arith1.fs

test-arith2.fs

test-array-basic.fs

test-assign1.fs

test-assign2.fs

test-assign3.fs

test-basic-struct.fs

test-crap.fs

test-duplicate-var-name.fs

test-easy.fs
```

test-failarg1.fs

test-failarg2.fs

test-failarg3.fs

test-for1.fs

test-for2.fs

test-for3.fs

test-hello.fs

test-if1.fs

test-if2.fs

test-if3.fs

test-mod1.fs

test-multiple-structs.fs

test-multirec.fs

test-ops1.fs

test-param1.fs

test-param2.fs

test-param3.fs

test-param4.fs

test-parl1.fs

test-parl2.fs

test-parl3.fs

test-parl4.fs

test-parl5.fs

test-parl6.fs

test-parlLinearSearch.fs

test-primefactor.fs

test-prlprimefactor.fs

test-rec.fs

test-var1.fs

test-while1.fs

test-while2.fs

test-while3.fs

## 7. Lessons Learned

- OCaml has a steep learning curve. I started the project somewhat late so it took me some time to grasp what was going on so I could do more than the very basics.
- As a CVN student, I felt it was harder to do to this pretty big project alone, especially when I was stuck and biting my head over some OCaml compiler error message oddity (funnily enough now ). Sometimes, this could suck huge amounts of time off, and it would have been a nice to run this over with someone else. Also felt it was hard for one person, to do everything, in terms of thinking of the report, learning this OCaml. So CVN students if you read this and have a choice get in a group. OCaml is not a language you want to get stuck on by yourself especially with limited help available online (that is if you can work with a person remotely).
- Try not to have such a grandiose vision like I had initially of working on both OpenMP and OpenMPI, which is infeasible.
- Functionally programming feels truly amenable to creating compilers, with its pattern matching, tools available around it, and the whole transformation paradigm that its associated with
- Overall, despite somewhat of a struggle it was well worth to see what functional programming was about, and what a compiler does, to truly get a deeper understanding of computer science.
-

## 8. APPENDIX