Civ

Michael Nguyen, Prateek Sinha, Yuchen Zeng, and Eli Bogom-Shanon

December 17, 2014

Contents

1 Introduction

1.1 What is Civ

Civ is a new language implemented by the authors as an academic project for Programming Languages and Translators class (COMS W4115) taught by Prof. Stephen A. Edwards.

1.2 Motivation

C is one of the most used languages in the world, however it is hard for the novice programmers to just learn the basic programming concepts given the myriad number of features and concepts associated with C. Thus for our project we decided to implement a subset of C which would provide our users an easy programming environment that enables them to quickly grasp the fundamental programming concepts such as control structures, data types, functions, etc.

1.3 Differences from C

Dynamic Arrays - All arrays are dynamic by default, all done by backend malloc. Memory is freed and reallocated automatically.

Automatic Garbage Collection - All mallocs are automally deallocated at the exit of a lexical scope. With these two points, memory management can largely be abstracted away from the user.

No pointers or addresses - Given these two differences, Civ has no usage of pointers or references, thus providing a level of safety for the beginning user.

2 Language Tutorial

2.1 Basics

Civ is a subset of C which means that it follows the same syntax as that of C language and it supports multiple features of C.

Civ programs are saved with ".mc" extension. The compiler takes this file and outputs C code provided there are no syntactical or semantic errors. This C code then can be saved to a output file and executed through GCC.

2.2 Keywords and Data Types

Civ has the following types:

int - signed integers.

float - signed floats, including scientific notation (e.g. 3.e-15).

char - standard ASCII characters, to include escape characters with '\

String - strings are supported, though they immediately get converted to character arrays.

2.3 Examples

Hello World

```
int main(){
    printf("Hello World!");
    return 0;
}
```

GCD

```
int main() {
  int a, b, t, gcd, lcm;
  int x = 4;
  int y = 18;
  a = x;
  b = y;

  while (b != 0) {
    t = b;
    b = a % b;
    a = t;
  }

  gcd = a;
  lcm = (x*y)/gcd;

  printf("Greatest common divisor of %d and %d = %d\n", x, y, gcd);
  printf("Least common multiple of %d and %d = %d\n", x, y, lcm);

  return 0;
}
```

Dynamic Arrays

```
/*Short tutorial on multidimensional arrays*/
int main(){

  /*declare an array without an initial size*/
  int a[];

  /*place elements in any cell without worrying about initializing them*/
  a[3] = 42;

  /*multidimensional arrays have the same approach*/
  int m[][][];
  m[1][2][3] = 123;

return 0;
}
```

```
/* Functions can return arrays and take arrays as arguments*/
int[] sample(int f[]){

/*if assigning one array to another, make sure to declare the array first*/
int x[];
x = f;
/*x now holds the same contents as f*/
int x[1] = 1;
int z = x[1];
        printf("%d \n", z);
/*make sure to always return a return value of the type you declared
     in the function signature*/
return x;
}
```

2.4 Compiling and Executing Civ

In-order to compile and execute a Civ program the user needs to save the file with a ".mc" extension within the root directory of where the Civ compiler is stored, e.g. /home/user/code/civ/test/civprogram.mc From there:

1 `./civ --gcc < \[path or regex\] > output.c`

civ takes in a regex or a path, meaning it can compile multiple files at once. This is a byproduct of its original use as a test suite, but works just as well. This will compile the output file through gcc and create an executable where the source code is, e.g.

```
/home/user/code/civ/test/civprogram.exe
```

2 From there, just execute the file, e.g.:

```
./home/user/code/civ/civprogram.exe
```

3 Language Reference Manual

3.1 Preface

This language reference manual describes the Civ language, developed by, Michael Nguyen, Prateek Sinha, Yuchen Zeng, and Eli Bogom- Shanon for Stephen Edwards's Programming Languages and Translators class (W4115). Given its similarity to the C language, this document closely follows an organizational precedent set by Brian Kernighan and Dennis Ritchie in their "The C Programming Language."

3.2 Introduction

Civ is a computer language based on C. However are a few major differences between Civ and C. Civ has no explicit usage of pointers in its syntax. This means that the symbol * is only used in Civ for exponentiation and multiplication functions. Because there are no explicit pointers, there are also no explicit

references using the & symbol. Civ also provides dynamically allocated arrays, whereas in C, native arrays are static in nature. While arrays created dynamically in C requires explicit calls to memory allocation functions, as well as the associated free calls, Civ handles memory allocation and garbage collection automatically. Civ is meant to provide a simplified version of C that enables a user to quickly grasp fundamental programming concepts, such as control structures, data types, functions, etc., without having to learn pointer arithmetic or memory management.

3.3 Lexical Conventions

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

Comments Comments are styled after the C multiline comments. A comment block begins with the characters /* and is terminated with */ . Civ does not provide nested comment support.

```
/* This is a comment in Civ */
/*
This is a multiline comment in Civ
*/
// Unlike C, this is not a valid comment
```

Identifers In Civ, an identifier is an alphanumeric sequence. Upper case and lower case letters are considered to be different in Civ.

Keywords The following identifiers are reserved for the use as keywords, and may not be used otherwise:

- int
- float
- char
- true
- false
- String
- if
- else
- for
- while
- break

- continue
- return
- void

There are also a few built in functions: *printf* and *maxArrayElement*. Printf() is used in the same manner as the standard I/O function in C. maxArrayElement is declared as

```
int maxArrayElement(type array[]);
```

Type can be int, char, or float. The function returns the number of elements between the start of the array and the last element in use - that is, the effective size of the array in number of elements.

Constants There are several kinds of constants, as follows:

Integer constants An integer constant is a sequence of digits.

Character constants A character constant is 1 character enclosed in single quotes " ' ". Within a character constant a single quote must be preceded by a back-slash "\". Certain non-graphic characters, and "\" itself, may be escaped by preceding them with a '\'.

Floating constants A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

Strings A string is a sequence of characters surrounded by double quotes " " ". A string has the type array-of-characters (see below) and refers to an area of storage initialized with the given characters. The compiler places a null byte (\0) at the end of each string so that programs which scan the string can find its end.

3.4 Types

Civ supports three fundamental types of objects: characters, integers, and floating-point numbers.

Characters (declared, and hereinafter called, char) are chosen from the ASCII set.

Integers (int) are represented in 16-bit 2's complement notation.

Floating points (float) quantities have magnitude in the range approximately 10 ± 38 or 0; their precision is 24 bits or about seven decimal digits.

Besides the three fundamental types there are classes of derived types constructed from the fundamental types in the following ways:

Arrays of objects.

Strings arrays of chars

Functions which return objects of a given type.

Conversions Unlike C, Civ generally does not allow type conversions, either implicitly or explicitly. The programmer is expected to treat a given object as the same type for the duration of the object's lifetime.

3.5 Objects and lvalues

An object is a manipulatable region of storage; an 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. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

3.6 Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators is summarized at the end of this section. Otherwise the order of evaluation of expressions is undefined.

Primary Expressions Primary expressions involving subscripting and function calls group left to right.

identifier An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration.

constant A decimal or floating constant is a primary expression. Its type is int in the first case and double in the second.

string A string is a primary expression. Its type is "array of char".

(expression) 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.

primary-expression (expression-list) A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning . . .", and the result of the function call is of type ". . . ". In preparing for the call to a function, a copy is made of each actual parameter; thus, almost all argument-passing in Civ is by value. However, the array type in Civ is actually passed by pointer in the compiled target C code. While this is not directly shown to the user, it should be noted that in Civ, primitive

data types are passed by value, while the aggregate type array is passed by reference.

Unary Operators Expressions with unary operators gruop right to left.

- expression The result is the negative of the expression, and has the same type. The type of the expression must be int or float.

! expression The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is int. This operator is applicable only to ints.

Multiplicative Operators The multiplicative operators *, /, and % group left-to-right.

expression * expression The binary * operator indicates multiplication. If both operands are of type int, the result is int; if both are type float, the result is float. If one is int and the other is float, the former is converted to float and float is returned.

expression / expression The binary / operator indicates division. The same type considerations as for multiplication apply.

expression % expression The binary % operator yields the remainder from the division of the first expression by the second. Both operands be int, and the result is int.

Additive Operators The additive operators + and - group left-to-right.

expression + expression The result is the sum of the expressions. If both operands are int, the result is int. If both are float, the result is float. If one is int and one is float, the former is converted to float and the result is float. No other type combinations are allowed.

expression - expression The result is the difference of the operands. The same type considerations as for + apply.

Relational operators The relational operators group left-to-right, but this fact is not very useful; "$a < b < c$" does not mean what it seems to.

expression < expression

expression > expression

expression <= expression

expression >= expression The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Operand conversion is exactly the same as for the +.

Equality operators

expression == expression

expression != expression The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus "a<b == c<d" is 1 whenever a<b and c<d have the same truth-value).

expression && expression The && operator returns 1 if both its operands are non-zero, 0 otherwise. Unlike &, && guarantees left-to-right evaluation. The operands need not have the same type, but each must have one of the fundamental types.

6.8 expression || expression The || operator returns 1 if either of its operands is non-zero, and 0 otherwise. Unlike | , || guarantees left-to-right evaluation. The operands need not have the same type, but each must have one of the fundamental types.

Assignment Operators There are a number of assignment operators, all of which group right-to-left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

3.7 Declarations

Declarations are used within function definitions to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

```
declaration:
    type-specifier declarator-list;
```

The declarators in the declarator-list contain the identifiers being declared. The type-specifier consists of one type-specifier.

Type specifiers The type-specifiers are:

```
type-specifier:
    int
    char
    float
    void
```

A type-specifier must be included in each declaration. The void type can only be declared as the return type of a function.

Declarators The declarator-list appearing in a declaration is a comma-separated sequence of declarators.

```
declarator-list:
    declarator
    declarator , declarator-list
```

The specifiers in the declaration indicate the type of the objects to which the declarators refer. Declarators have the syntax:

```
declarator:
    identifier
    declarator ( )
    declarator [ constant-expression ]
    ( declarator )
```

The grouping in this definition is the same as in expressions

Meaning of Declarators Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type. Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

If a declarator has the form

```
D( )
```

then the contained identifier has the type "function returning ...", where ". . . " is the type which the identifier would have had if the declarator had been simply D.

```
D[ ]
```

It is valid to use this declarator without a constant expression. Such a declarator makes the contained identifier have type "array." If the unadorned declarator D would specify a nonarray of type ". . .", then the declarator "$D[]$" yields a 1-dimensional dynamic array of objects of type ". . .". If the unadorned declarator D would specify an n-dimensional array with rank $i_1, i_2, ..., i_n$, then the declarator "$D[i_n + 1]$" yields an (n +1)-dimensional array with rank $i_1, i_2, ..., i_n, i_{n+1}$.

An array may be constructed from one of the basic types.

Finally, parentheses in declarators do not alter the type of the contained identifier except insofar as they alter the binding of the components of the declarator.

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return functions; there are also no arrays of functions. Here, Civ is slightly more restrictive than C, as in C some of these restrictions may be circumvented through use of pointers.

3.8 Statements

Except as indicated, statements are executed in sequence.

Expression statement Most statements are expression statements, which have the form:

```
expression ;
```

Usually expression statements are assignments or function calls.

Compound statement So that several statements can be used where one is expected, the compound statement is provided:

```
compound-statement:
    { statement-list }

statement-list:
    statement
    statement statement-list
```

Conditional statement The two forms of the conditional statement are:

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

In both cases the expression is evaluated and if it is non-zero, the first sub-statement is executed. In the second case, the second sub-statement is executed if the expression is 0. As usual the "else" ambiguity is resolved by connecting an else with the last encountered else-less if.

While statement The while statement has the form:

```
while ( expression ) statement
```

The sub-statement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

For statement The for statement has the form:

```
for ( expression-1 ; expression-2 ; expression-3 ) statement
```

This statement is equivalent to:

```
expression-1;
while ( expression-2 ) {
    statement
    expression-3 ;
}
```

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration.

Break statement The statement

```
break ;
```

causes termination of the smallest enclosing while, do, or for statement; control passes to the statement following the terminated statement.

Continue statement The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing while, do, or for statement; that is to the end of the loop.

Return statement A function returns to its caller by means of the return statement, which has one of the forms

```
return ;
return ( expression ) ;
```

In the first case no value is returned. In the second case, the value of the expression is returned to the caller of the function. The expression must evaluate to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value. There are some exceptions to this. A return expression cannot of type element-of-array.

```
return( a[0] );
```

is not a valid statement in Civ. The value must be assigned outside of the return statement, then passed in.

3.9 Scope rules

Civ, unlike C, has a strictly lexical scope. Civ is a block-structured language. The lexical scope of names declared at the head of functions (either as formal parameters or in the declarations heading the statements constituting the function itself) is the body of the function. It is an error to re-declare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

3.10 Arrays

Every time an identifier of array type appears in an expression, it is treated as array-of-(type at declaration). Because of this, arrays are not lvalues. The subscript operator [] is interpreted in such a way that if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. A consistent rule is followed in the case of multi-dimensional arrays. If E is an n-dimensional array of rank $i, j, ..., k$, then E appearing in an expression evaluates from array-of-array of (n-1)-dimensional array with rank j , ... ,k to the type held in dimension k.

3.11 Example code

The following code executed a bubblesort on an array of integers. It showcases the use of the dynamic array datatype in Civ, an shows a brief example of using the formatted print option, which is modeled after C's own printf().

```
void bubblesort(int t[]){
    int i,j;
    int n;
    n = maxArrayElement(t) + 1;
    for(i = 1; i < n; i = i + 1){
        for(j = 0; j < n - i - 1; j = j + 1){
            if(t[j] > t[j + 1]){
                int a = t[j];
                int b = t[j + 1];
                int temp = t[j];
                t[j] = t[j + 1];
                t[j + 1] = temp;
                printf("SWAPPING: %d %d \n",a,b);
            }
        }
    }
return;
}

void main(){
    printf("Bubblesort \n");
    int g[];
    int z;
    for(z = 10; z > 0; z = z - 1){
    g[10 - z] = z;
    }
    for(z = 0; z < 10; z = z + 1){
    int temp = g[z];
    printf("%d ", temp);
    }
    bubblesort(g);
    printf("Sorted! \n");
    for(z = 0; z < 10; z = z + 1){
        int temp = g[z];
        printf("%d \n", temp);
    }
    return;
}
```

4 Project Plan

4.1 Planning, Development, and Testing

Planning We started with weekly meetings to discuss the features we wanted to implement in our language. At the beginning we had the idea of implementing a distributed language with the target of achieving the functionality of Map Reduce. However, since none of the team members had in-depth knowledge of the concept, we realized that it would be very difficult to come up with the solutions to the problems we would be facing while development thus, we consulted the instructor and by the end of September we started working towards the idea of implementing Civ.

We kept the same schedule, meeting every week on Sunday to discuss and kept meeting our TA Vaibhav on a regular basis to resolve various issues that came up while implementing Civ.

Development For development we used Git as our version control system and checked out work into a remote repository on GitHub. Just by happenstance, we ended up adopting a wave strategy of implementation, as we weren't sure how to allocate our workload, and our project leader had medical issues. The first wave was Mike, who wanted to implement as much of a working language as possible, even if it meant C to C. He developed the grammar, ast generation, and code generation the first week of December. Yuchen was the second wave who implemented the semantic analyzer and the SAST, and Mike worked with him to merge it into the pipeline. Eli came in as third wave and implemented the core pieces of our language, with the dynamic arrays and garbage collection during the last week. During this entire process, eveyone was either writing test cases, building auxiliary tools, or helping any way they can to support the current wave.

Testing A test suite was written about halfway through that allowed rapid testing and feedback of our code. It supported both expected passing and expected failing cases to highlight false positives or true negatives, and allowed quick isolation of where the errors were. This test suite later turned into an extension of the compiler itself.

4.2 Style Guide

Our overall guiding point was to minimize code redundancy, so both Mike and Yuchen wrote a lot of auxiliary functions (especially for code generation) that converted variou stypes into strings or other types depending on the situtation. The grammar was also meant to minimize redundancy, and utilized a lot of recursion like most grammars do to account for strange cases. In terms of actual practice, we had the following rules:

- Never ever push to origin master.
- Never ever push unless the test suite runs and things compile. Commits are fine as stopping points, but there should never be a reason to rollback.

- Code should be documented, especially at the beginning of newly introduced functions in OCaml.
- Small changes that don't contribute to a huge portion of the language can be marked with TODO:
- Everyone owns a stake - Mike owns scanner.mll, parser.mly, and ast.ml, and Yuchen owns sast.ml, semantic,ml, and ccompilesast.ml.
- When developing features, make both passing and failing test cases, and put them in the test directory in the appopriate slot.
- Tests in a feature should be incremental, i.e. a test for single array declaration, then nested array, then double nested array, etc.
- Update the README with your contribution at the end of the night, marking UNTESTED for future testing or for someone else to write tests.
- When generating C, use camelcase for our library.
- Python should follow PEP8 style guide.

4.3 Project Timeline

Our ideal scenario would have ended up looking something like this:

September - Get a fully fleshed out idea of what our language would look like and do. This includes writing basic benchmarks for compilation that incorporate more and more features of the language.

October - Develop and test the scanner and parser. Print out the AST and hand verify to confirm that it is working as intended. Continue developing the language in terms of its scope and core features.

November - Half of the team work on pre-emptive code generation, and the other half on a semantic analyzer. As it is likely code generation will finish first, have them test/debug/even out the rest of the compiler.

December - Fully integrate the semantic analyzer's SAST with code generation. At this point, the scanner, parser should be fully tested and complete, and the code generation should be a easily modified to work with the new SAST as opposed to the AST. Because it's a bad idea to have four people work on the same file, some of the people will be working on the documentation and final report.

4.4 Project Log

September - November We had a lot of talking and very little coding done through these months. By the end of it, we had a rough idea of the language, but we had no idea how to go about actually implementing it, especially regarding dynamic arrays or garbage collection.

December This is where development began.

12/16/14
Yuchen/Eli –
* Re-implemented dynamic arrays at the lsat minute

Prateek –
* Finished slides/presentation and report

12/15/14
Mike –
* Fixed float parsing
* Test suite now fully gcc's
* Added in continue/break
* Varchaining fully implemented
* Fixed some code generation (For/Call/If/While)

Yuchen/Eli –
* Fully implemented and tested dynamic arrays/garbage collection

12/14/14
Mike –
* Strings are in e.g. char x[] = "test"; -- see SAssign in ast.ml
* String declarations in char x[] = now string * string * string list
* For loop's last argument is now stmt list as opposed to stmt
* If's last argument is now stmt list as opposed to stmt
* Escape characters are in treated as chars of max size 2.
* Variable declaration chaining in e.g. int x,y; - see VDecllist in ast.ml
* INCR/DECR added under expressions - Will add a type under EXPR for it later

Yuchen –
* Dynamic Arrays are in
* Add string in sast, semantic and ccompilesast
* Fixed a few test cases

12/13/14
Mike/Yuchen –
* Arrays now have their own ID type
* Arrays can be used to describe formal arguments
* Changed all iliterals into expressions that hopefully get resolved
* Added in GCC to test suite

Yuchen/Eli
* Code Generation of Static/Dynamic Arrays

Eli –
* Added in more testing of the dynamic array header file

12/12/14
Mike –
* Consolidated pipeline
* Fixed all tests cases that use single line comments
* Added more static array test cases
* Added more features to test suite
* Static arrays fully functional
* Dynamic array declarations in

Eli –
* Added set of test cases for dynamic arrays
* Single dynamic array C header up
* Prototype for C automatic garbage collction up

Yuchen –

* Consolidated pipeline
* Add 'Printlist' in sast
* Add semantic checking for static array declaration
* Worked more miracles

Prateek -
* Doubled test cases to ~100
* Reorganized all test cases into PASS/-feature and FAIL/-feature folders to test individual features

12/11/14
Eli -
* Split up all test cases into incremental

Yuchen -
* Worked miracles in semantic analyzer
* Add 'Array' and 'Print' in sast
* Make the compile using sast work

12/10/14
Mike -
* Redid tester = fully operational pending further features
* I hate recursive data types - can't figure out how to do array
* Printf now works with (str,args);

Yuchen
* Add return type checking
* Write the function convert program in ast to program in sast and merge sast into the pipeline

12/9/14
Eli -
* Added more tests cases to account for arrays
* Progress on dynamic arrays and automatic garbage collection

Yuchen -
* Most problems about scope are solved and tested (Scope for 'While' and 'For', scope for global environment, scope for multi functions, scope for formals)
* Add test cases for scope and multi functions

###12/8/14###
Yuchen -
* Problem with 'call' is solved and tested. Function's name and type, arguments' number and types are checked.
* Add semantic checking and ast-sast converting for 'Call', 'Return'

12/6/14
Mike -
* Nested Arrays are in

Eli -
* Strategy for implementing pointerless C done

Yuchen -
* Worked towards putting SAST between AST -> CCompile

Prateek -
* Rewrote python test script

12/6/14
Yuchen -

```
* Add semantic checking and ast-sast converting for 'While', 'For', 'VDecl'
  UNTESTED - semantic checking and ast-sast converting for 'While', 'For'
* Add semantic checking: if there is id confict when initialing new vairable
in both 'VDecl' and 'NAssign'

12/5/14
Yuchen -
* Added Types.ml, Sast.ml
* UNTESTED - Began work on Semantic; implemented: utility functions for
AST traversal,
scoping environments, equality tests, type checking, type requirements,
environmeny var/func checks

12/4/14
Mike -
* IN PROGRESS - Added in array optionals that come after ID Token
* Adjusted TYPE ID ASSIGN expr to statements. CONSIDER MOVING BACK TO EXPR FOR CHAINED.
* Global declarations are now ALWAYS TYPE ID ASSIGN LITERAL.
* Arithmetic operations working as expected
* UNTESTED - float literals - currently viewed as strings
* UNTESTED/OPTIONAL - Added break,const,continue,extern,float,static
* UNTESTED - Added increment/decrement, NOT ADDED TO GRAMMAR YET
* More test cases

Eli -
* Created more in-depth test cases

Prateek -
* Added python test script

12/3/14
Mike -
* Var Declaration can occur anywhere, e.g. int x; now works like c99 standard
* All grammar rules accounted for with two conflicts
* Formal arguments take type now
* Print accounted for
* Added three print test cases in tests/ps/
* Global variable INITIALIZATION in (seems useless)
* UNTESTED - Strings added to lexer/parser/compiler
* UNTESTED - Chars added, same as strings

12/2/14
Mike -
* Code generation up and running!
* Basic formatting to do basic C code up.
* UNTESTED - Added print statement to scanner and grammar
* UNTESTED - Differentiated new variable declaration AND assignment
* Removed all old code e.g. bytecode/compile (now ccompile), etc.
* Removed all old test cases and modified Makefile for new environment


12/1/14
Mike -
* Started over from scratch
* Added in type declarations for functions
* Added in variable declaration and assignment of expression
* Added ccompile/ccode.ml to be used for actual compilation
* Microc now has a -C flag that is used for actual ccompile.translate
* Codegeneration has begun - need to adjust formatting and account for type_decl string forma
* Mikhail helped
```

4.5 Software Environment

Operating Systems Windows, Linux, Mac OS

Core Language OCaml 4.01.0

Scripting Python 2.7

C Compiler GCC 4.6

4.6 Roles and Responsibilities

Eli Bogom-Shanon - Core Language Designer

Michael Nguyen - Project Lead, Environment/Git Master, Test Suite Developer, Grammar Developer

Prateek Sinha - Documentation and Test Case maker

Yuchen Zeng - Core Developer, Semantic Analysis and Code Generation Developer

5 Architecture

Civ uses a textbook setup. The sequence is as follows:

Scanner - The scanner tokenizes an input string into a set of tokens. Any substring that isn't recognized implies it is not a valid program, so the program is immediately rejected.

Parser - The parser takes a set of tokens and builds an Abstract Syntax Tree from it using pre-defined context free grammar.

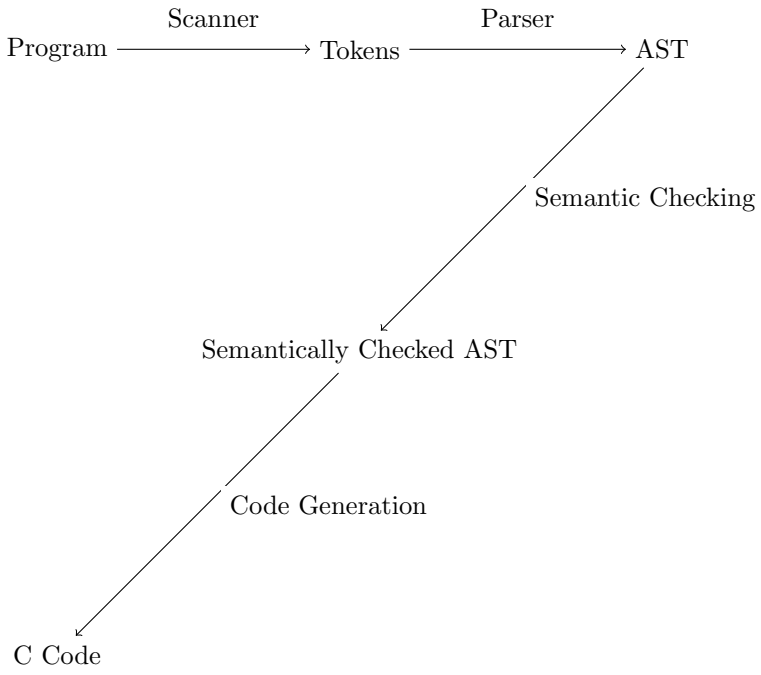
Semantic Analyzer - The analyzer takes an AST and does semantic checking to make sure the abstract syntax tree 'makes sense'. In our case, it outputs near-C equivalent AST.

Code Generator/Compiler - This takes in the checked AST and outputs the actual C code, using a roughly one to one mapping.

6 Testing

Testing was one of the most significant part of our project; at every step we wrote new test cases to ensure that the newly developed parts worked as desired. The test suite for this project consists of an automated script written in python and about 100 test cases that we wrote to test different features of our language. The test suite has a plethora of flags and a helpfile using:

```
./civ -h
```



6.1 Automated Test Script

We wrote a python script to automate the process of testing our components. The script takes a directory of tests and runs each test case inside that directory. In other words it takes each file compiles them through civ and outputs a C file which it then runs through GCC. For each test case it specifies whether it failed or not and if failed it prints out whether it failed at civ or GCC. It also prints out the first line of each test case which says what the test case was supposed to do.

6.2 Test Suite Code

```
from subprocess import call, Popen, STDOUT, PIPE
from sys import exit

pt = lambda x,y: print("\033[%sm%s\033[0m" % (30 + x,y)) # Where 0 <= x <= 7

parser = argparse.ArgumentParser(description="Super testing suite for MilliC")
parser.add_argument('directory', help="Which test directory to use e.g. 'base'. Use '.' for all tests")
parser.add_argument('--nomillic', help='Disables makefile error messages', action="store_true")
parser.add_argument('--gcc', help="Does full compile but no running", action="store_true")
parser.add_argument('--noclean', help='Disable cleanup post run', action="store_true")
parser.add_argument('--nopass', help="Disables showing passing test cases", action='store_true')
parser.add_argument('--showpass', help="Shows passing test cases", action='store_true')
parser.add_argument('--nofail', help="Disables showing failing test cases", action='store_true')
parser.add_argument('--showfail', help="Shows failing test cases", action='store_true')
parser.add_argument('--suppress', help="Only shows gcc messages" ,action='store_true')
flags = parser.parse_args()
microcpath = os.getcwd() + "/microc"
FAIL,PASS,TOTALTESTS,TESTING = 0,0,0,0
CFAIL, CPASS, CTOTALTESTS, CTESTING = 0,0,0,0

def getTestPaths():
    global TOTALTESTS,TESTING
    allPaths = []
```

```
    current = os.getcwd()
    print(current)
    for tup in os.walk('./tests'):
        for test in tup[2]:
            allPaths.append(current + tup[0][1:] + '/' + test)
    allPaths = filter(lambda x: x[-2:] == 'mc', allPaths)
    TOTALTESTS = len(allPaths)
    allPaths = filter(lambda x: flags.directory in x, allPaths)
    TESTING = len(allPaths)
    return allPaths

def setup():
    show = False if flags.nomillic else True
    try:
        c = call(['make'])
        if c != 0:
            pt(1, "Makefile failed: Error Code %s" % c)
            exit()
    except Exception as e:
        pt(1, "Makefile failed:%s" % e)

def writeC(outfile,code):
    with open(outfile, "w") as f:
        f.write(code)

def test(filepath):
    outfile = filepath[:-3] + ".c"
    outc = filepath[:-3] + ".exe"
    filebase = filepath[:filepath.rfind('/') + 1]
    code = open(filepath).read().encode('ascii','ignore')
    global FAIL,PASS

    try:
        p = Popen([microcpath,'-SC'], stdin = PIPE, stdout = PIPE, stderr = STDOUT)
        out,err = p.communicate(input=code)
        if "error" in out and not flags.nofail:
            pt(1,"%s\n%s" % (filepath,out))
            if flags.showfail and not flags.suppress:  pt(3, code)
            FAIL += 1
            return
        elif not flags.nopass:
            pt(2, "%s PASSED" % filepath)
            if flags.showpass and not flags.suppress: pt(5,out)
            PASS += 1
            writeC(outfile,out)
    except Exception as e:
        pt(3, "Failed MilliC %s:%s" % (filepath,e))

    if not flags.gcc : return

    try:
        print(outfile)
        Popen(['gcc', "-o" + outc, "-std=c99", "-Iarrays", outfile],stdout=PIPE)
    except Exception as e:
        pt(3, "Failed GCC %s:%s" % (filepath,e))

if __name__ == "__main__":
    setup()
    for t in getTestPaths():
        test(t)
    if not flags.noclean:
        call(['make','clean'])
    pt(4,"Testing %s / %s Total" % (TESTING,TOTALTESTS))
    pt(4,"%s FAILED %s PASSED" % (FAIL,PASS))
    pt(4,"%s GCC FAILED %s GCC PASSED" % (CFAIL,CPASS))
```

6.3 Test Cases

The test cases that we created were classified as Pass and Fail. They were further classified into features that we were testing like conditional tests, control tests,

etc. In the end the criteria was that all the test cases in pass should compile successfully in civ and generate a proper C code which upon execution through GCC gives the expected output. Similarly the cases in Fail were supposed to fail when compiled through civ for either syntactical errors or semantic errors.

6.4 Test Phase

We had roughly two "phases" for our tests. The first was continually building the grammar and making sure everything parsed correctly, and once the full dynamic arrays and garbage collection was in, we started making sure code generation complied with the GCC C99 standard.

7 Lessons Learned

Mike Project leadership: In industry, everyone has a specialization and/or has a specific role that contributes to a whole. This is why there is a team breakdown of roles. Do not under any circumstance let it become nebulous with people having their hands in different aspects of the project. Make everyone an absolute dictator of their domain, and get everyone to be really aggressive about doing their work. Keep a work log - it becomes very evident who works and who doesn't, and the moment deadweight is detected, let the person know, and then cut them off.

Grammar design: Minimize redundancy, and use lots of self-referential grammar rules. You don't want to create cases that require one specific rule - it will lead to scrambling to consider of all the edge cases, and hours spent testing every single possible configuration.

Testing: Create a full test suite that is modular and takes in configurations. It will make life a lot easier to test all 50 - 100 test cases in one command line argument, with different options to display failing case code or running other unix commands on the object code. Minimizing the feedback time from compilation time shortens the development cycle.

Work allocation and strategy: While the 'thread' idea is a good idea, where you try to get one small aspect of your code to compile and run, a 'wave' idea worked out better for us. I was first wave in designing the grammar, AST, and code generation, and then the 'second' wave moved who worked on the semantic analyzer and the SAST generation AS I was fixing/upgrading. It allowed both of us to work at maximum productivity, because I gave him a foundation to start. The third wave moved in shortly after with him working on the 'centerpiece' of our language with dynamic arrays and garbage collection. This also means that peoples' workloads peaks at different places, allowing people with spare time to work on other issues, like testing and helping each other out. The alternative to this is everyone sitting around one coder, kind of like pair programming. That was a terrible strategy we used for the longest time, because it always ended up in people arguing and bickering over trivial details.

Advice: The earlier you work the faster you realize how painful this project can be. Get each person to have responsibility for something. Keep a

worklog to keep track of who is working and who isn't. Pair programming is great, but group programming is a waste of time.

Yuchen Semantic Analysis: There are a lot of things to check for semantic even we have correct AST. We have to traverse the AST using DFS to check the semantic of each literal, each ID, each expression and each statement. Unlike the lexical and syntax checking, we need to know the scope, the environment when we check everything. This should be organized and recorded carefully from the beginning of the checking process. Things should be thought clearly before the codes are written down, it is the way that makes the development process of semantic analysis more efficient.

Coding in Ocaml: Coding in Ocaml is a special experience. It is not like the language that I were familiar with. We have to do a lot of recursions when coding in Ocaml. But it also makes the coding experience very interesting. We were not tediously moving codes from one place to another. Every line of the codes were thought carefully and a short codes can made what you want. At the beginning, the coding process is slow. But once I got used to the style of Ocaml coding, the process got more and more efficient. It was a good training of my thought of coding.

Teamwork: I worked with Mike on the converting of AST to SAST and worked with Eli on the dynamic arrays and garbage collection. Once Mike made some changes on parser and AST, I had to follow him to make corresponding changes in SAST and semantic checking. Eli provided the C methods that could be used in dynamic arrays and garbage collection, and I should make the generated codes use these methods in the right way. Such experience gave me the training of how to work with other people as a team.

Advice: Think early about how to implement what you want. Make some tests about the prototype. Try to be clear of what the work will be like when starting to write codes.

Eli Management and organization are really important to get right. While a project should be a collaboration, a design major friend once told me he can always tell when a project was designed by a committee rather than a lead designer because it is a mix of possibly good ideas all executed poorly. It is important to find a balance between collaboration and leadership. Swing too far either way and the project goes South quickly. It is ok to be flexible about team roles. If it seems like someone is uncomfortable in a certain role, let them know that they can switch roles. If it seems like someone is unable to act in their capacity in a certain roll, be proactive in approaching them about the problem. Letting someone continue to perform a roll in the project that they are unable to do is a detriment to them as well as to the rest of the team. This is especially important for the management role, as if the manager is unable to perform their duties, it may not be clear where other problems in the project lie.

As a second thing, while this is not really a new lesson, it bears repeating that the early one starts on a project, the better things will turn out. Especially in testing, there are times where something that you may think is a tiny bug that will take 10 minutes to fix ends up taking 10 hours.

Testing is a very important and unbelievably time consuming step, and should be considered a very major portion of the project.

Prateek While working on this project I realized how important various aspects of working in a team are. Strict time-line, work allocation, proper communication and team management everything affects the final product. During the early stages of development we tried to code in pairs and it didn't work so well for us. So even though meeting regularly is important it is even more important to allocate the work properly. I also realized that for a project of this size it is important that the team members should take initiatives and bring more and more ideas to the table.

My advice for the future teams would be to start implementation as early as possible. Ocaml is a difficult language and it takes time to get acquainted with it. While writing the test cases even though the code blocks are good to test the overall system, small test cases help you realize where exactly the error is hence it is better to write incremental test cases.

8 Appendix

See tarfile for code

References

[1] B. W. Kernighan and D.M. Ritchie *The C Programming Language* Prentice Hall, Englewood Cliffs New Jersey 1978