Cπ

Programming Languages and Translators, COMS4115

Edward Garcia (ewg2115), Niket Kandya (nk2531),

Naveen Revanna (nr2443), Sean Yeh (smy2112)

December 20,2013

Table of Contents

1 INTRODUCTION

C is the lingua franca of the computer world. It is a general purpose programming language and often used in systems level programming. In this project we will be implementing Cπ (Cpi) which is a subset of the C language. It will be designed to compile to ARM V6 assembly with the target platform being the Raspberry Pi (RPi) . Cπ will use the GNU assembler(as/gas), linker(ld) and linaro cross toolchain (gcc-linaro-arm-linux-gnueabihf-raspbian) for assembly to binary code generation on the RPi.

1.1 KEY FEATURES

Cπwill be an easy language to learn for those familiar with ANSI C. The generated ARM V6 assembly will be completely unoptimized.

1.1.1 KEYWORDS

Cπ will contain the following keywords

int	char	void	struct	NULL
while	if	else	for	void

1.1.2 PRIMITIVE DATA TYPES

```
int:        32 bit integers
char:       8 bit character
int*:       32 bit pointer to an integer value
char*:      32 bit pointer to a character value
void*:      32 bit pointer to a castable value
```

1.1.3 AGGREGATE DATA TYPES

These will be defined/declared as in C collecting primitive datatypes.

```
array
structure
```

1.1.4 OPERATORS

Operator precedence will follow standard orders of operation and will mimic ANSI C.

```
,     For arrays, structure definition, separate expressions
```

```
[]    Array indexing
*     Unary * for pointer dereferencing
.     For accessing structure members through a structure variable
->    For accessing structure members through a structure pointer
&     Returns address of a datatype
==    Returns an int (1 if equality holds, 0 otherwise)
!=    Returns an int (0 if equality holds, 1 otherwise)
&&    Logical AND operator
||    Logical OR operator
=     Assignment operator
>     greater than operator
<     less than operator
>=    greater than operator
<=    less than operator
+     Addition Operator
-     Subtraction Operator
*     Multiplication Operator
/     Division Operator
```

1.2 UNSUPPORTED FEATURES FROM ANSI C:

- Floating point variables and operations - no double and float
- short and long integers
- Unsigned integers
- break and continue
- Enums
- sizeof and Ternary operators
- Increment and decrement operators.
- Const
- If else statement
- Global declarations
- `do-while` and `switch` statements.
- Storage class specifiers - auto, register, volatile static and extern.
- Multi-file compilation and linkage.
- Preprocessing - no # directives.
- Function pointers.
- Function inlining.
- Static and volatile function.
- Variable function arguments - Ellipsis (...)
- No type casting

2 Language Tutorial

Given the widespread popularity of the C programming language, this tutorial will focus on aspects of Cpi which differ from c.

2.1 Programming in Cpi compared to ANSI C

2.1.1 *Scoping Rules*

The biggest deviation from ANSI C is the new scoping rules which Cpi implements. A program in Cpi is defined as a series of structure definition and function declaration. Therefore, there are no global variables declarations allowed in Cpi. In addition, a function is defined as a series of variable declarations followed by statement lists. Therefore, variables cannot be declared after the first statement in a function in Cpi. This also means that variable assignments must follow variable declarations and an assignment and declaration cannot happen on a single line.

2.1.2 *Stricter Type Checking Rules*

In ANSI C, certain type checking rules between different pointers and integers throw warnings to users while still compiling the source code unless certain flags are enabled. In Cpi, while these type checking rules are the same, an error is thrown and the compiler exits without compiling the program.

2.1.3 *Variable Sized Array Declarations*

Within a function in Cpi, it is possible to declare an array of variable size assuming the array size was passed at a function parameter. Arrays declared in this manner are allocated on the stack and is explained in detail in section 5.4. An example program is given in section __.

2.1.4 *No else if, Switch Statements or increment/decrement operators*

While these statements and expressions are used often, we chose to focus on implementing core C features first and return to these features if time permitted. All of these statements and operators functionality can be implemented in other ways in Cpi. A list of workaround is given in section 2.2.

2.2 Common Substitutes to Implementing C features in Cpi

2.2.1 *Global Variables*

While global variables and structures are not implemented, it is possible to expand the scope of a variable or structure to another function by passing pointers in function arguments. Cpi supports the use of malloc and data structures can be dynamically allocated, passed, and freed.

2.2.2 *Declaring and Assigning Structures, Variables and Pointers*

Variable, pointer, and structure declarations must happen separately from assignments within a function with declaration occurring first. The following code provides an invalid and valid declaration/assignment example in Cpi.

```
/* ********************************************************
 *    Valid declarations of variable and assignments on stack
 *
 *********************************************************/
void fun(){
     struct s{             //invalid definition of struct in
          int a;           //function
          int b;
     };

     int a[2] = {1, 2};  //invalid assignment of array during
                          //declaration
     char b[] = "Hi";    //invalid assignment of array during
                          //declaration
     int c = 1;           //invalid assignment of int during
                          //declaration
     int *p = &c;         //invalid pointer assignment during
                          //declaration
     c = 3;               //valid
     int d;               //invalid declaration of int after variable
                          //declarations


     while (1){
          int i;          //invalid declaration of int after
                         //variable declarations

     }

}
/* ********************************************************
 *    Valid declarations of variable and assignments on stack
 *
 *********************************************************/
struct s{
     int a;
     int b;
};

void fun(){
     int a[2];
     char b
     int c
     int *p
     int d;
     int i;
```

```
        int[0] = 1;
        int[1] = 2;
        c = 1;
        p = &c;

        while (1){


        }

}
```

2.2.3 *Else if and Switch Statements*

The code below shows a common usage scenario for else if and switch statements and a substitute in Cpi for achieving the same results.

<pre>switch(i)
{
    case -1:
        n= n + 1;
        break;
    case 0 :
        z= z + 1;
        break;
    case 1 :
        p = p + 1;
        break;
    default:
      i = i + 1;
}</pre>
	<pre>if (i==-1){
	      n = n + 1;
	}else if (i == 0){
	      z = z + 1;
	}else if(i == 1){
	      p = p + 1;
	}else{
	      i = i + 1;
	}</pre>
		<pre>if (i == -1){
		     n = n + 1;
		} else {if (i == 2){
		      z = z + 1;
		} else {if (i == 3){
		      p = p + 1;
		} else {
		      i = i + 1;
		}}}</pre>

TABLE 1. SUBSTITUTE FOR ELSIF AND SWITCH STATEMENTS

2.2.4 *Increment and Decrement*

Increment and decrement operations can be implemented with the following expressions

```
int k;
k = k +1; //increment
k = k -1; //decrement
```

3 Language Reference Manual

Lexical conventions

There are six kinds of tokens: identifiers, keywords, constants, 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. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

Comments

The characters /* introduce a comment, which terminates with the characters */. The character // introduce a comment which terminates upon reaching the end of a line

Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore "_" counts as alphabetic. Upper and lower case letters are considered different. No more than the first eight characters are significant, and only the first seven for external identifiers.

Constants

There are several kinds of constants, as follows:

Integer constants

An integer constant is a sequence of digits.

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. In a string, the character " " " must be preceded by a "\" ; in addition, the same escapes as described for character constants may be used.

Data Type Combinations

There is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

arrays of objects of most types;

functions which return objects of a given type;

pointers to objects of a given type;

structures containing objects of various types.

In general these methods of constructing objects can be applied recursively.

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. There are operators which yield lvalues: for example, if E is an expression of pointer type, then *E is an lvalue expression referring to the object to which E points. 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.

CONVERSIONS

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This section explains the result to be expected from such conversions.

CHARACTERS AND INTEGERS

A char object may be used anywhere an int may be. In all cases the char is converted to an int by propagating its sign through the upper 8 bits of the resultant integer. This is consistent with the two’s complement representation used for both characters and integers. (However, the sign-propagation feature disappears in other implementations.)

POINTERS AND INTEGERS

Integers and pointers may be added and compared; in such a case the int is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case the result is converted to an integer as specified in the discussion of the subtraction operator.

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 in an appendix.

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. However, if the type of the identifier is ‘‘array of . . .’’, then the value of the identifier expression is a pointer to the first object in the array, and the type of the expression is ‘‘pointer to . . .’’. Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared ‘‘function returning . . .’’, when used except in the function-name position of a call, is converted to ‘‘pointer to function returning . . .’’.

CONSTANT

A decimal or character constant is a primary expression.

STRING

A string is a primary expression. Its type is originally ‘‘array of char’’; but following the same rule as identifiers, this is modified to ‘‘pointer to char’’ and the result is a pointer to the first character in the string.

(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]

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type ‘‘pointer to . . .’’, the subscript expression is int, and the type of the result is ‘‘ . . . ’’. The expression ‘‘E1[E2]’’ is identical (by definition) to ‘‘* ((E1) + (E2)) ’’.

PRIMARY-EXPRESSION (EXPRESSION-LISTOPT)

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 ‘‘ . . . ’’. As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

In preparing for the call to a function, a copy is made of each actual parameter; thus, all argument-passing in Cpi is strictly by value. A function may change the values of its formal parameters, but these changes cannot possibly affect the values of the actual parameters. On the other hand, it is perfectly possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. Recursive calls to any function are permissible.

PRIMARY-LVALUE . MEMBER-OF-STRUCTURE

An lvalue expression followed by a dot followed by the name of a member of a structure is a primary expression. The object referred to by the lvalue is assumed to have the same form as the structure containing the structure member. The result of the expression is an lvalue appropriately offset from the origin of the given lvalue whose type is that of the named structure member. The given lvalue is not required to have any particular type.

PRIMARY-EXPRESSION −> MEMBER-OF-STRUCTURE

The primary-expression is assumed to be a pointer which points to an object of the same form as the structure of which the member-of-structure is a part. The result is an lvalue appropriately offset from the origin of the pointed-to structure whose type is that of the named structure member. The type of the primary-expression need not in fact be pointer; it is sufficient that it be a pointer, character, or integer.

Except for the relaxation of the requirement that E1 be of pointer type, the expression ‘‘E1−>MOS’’ is exactly equivalent to ‘‘(*E1).MOS’’.

UNARY OPERATORS

Expressions with unary operators group right-to-left.

** EXPRESSION*

The unary * operator means indirection: the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is ‘‘pointer to . . .’’, the type of the result is ‘‘ . . . ’’.

& LVALUE-EXPRESSION

The result of the unary & operator is a pointer to the object referred to by the lvalue-expression. If the type of the lvalue-expression is ‘‘ . . . ’’, the type of the result is ‘‘pointer to . . .’’.

- EXPRESSION

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

MULTIPLICATIVE OPERATORS

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

*EXPRESSION * EXPRESSION*

The binary * operator indicates multiplication. If both operands are int or char, the result is int; No other combinations are allowed.

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 must be int or char, and the result is int. In the current implementation, the remainder has the same sign as the dividend.

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 or char, the result is int. If an int or char is added to a pointer, the former is converted by multiplying it by the length of the object to which the pointer points and the result is a pointer of the same type as the original pointer. Thus if P is a pointer to an object, the expression ''P+1'' is a pointer to another object of the same type as the first and immediately following it in storage. No other type combinations are allowed.

EXPRESSION − EXPRESSION

The result is the difference of the operands. If both operands are int, char the same type considerations as for + apply. If an int or char is subtracted from a pointer, the former is converted in the same way as explained under + above. If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an int representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object-length.

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 + operator except that pointers of any kind may be compared; the result in this case depends on the relative locations in storage of the pointed-to objects. It does not seem to be very meaningful to compare pointers with integers other than 0.

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; moreover the second operand is not evaluated if the first operand is 0. The operands need not have the same type, but each must have one of the fundamental types or be a pointer.

EXPRESSION ||EXPRESSION

The || operator returns 1 if both either operands are non-zero, 0 otherwise. || guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 1. The operands need not have the same type, but each must have one of the fundamental types or be a pointer.

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.

LVALUE = EXPRESSION

The value of the expression replaces that of the object referred to by the lvalue. The operands need not have the same type, but both must be int, char, or pointer. If neither operand is a pointer, the assignment takes place as expected, possibly preceded by conversion of the expression on the right. When both operands are int or pointers of any kind, no conversion ever takes place; the value of the expression is simply stored into the object referred to by the lvalue. Thus it is possible to generate pointers which will cause addressing exceptions when used.

EXPRESSION , EXPRESSION

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left-to-right. It should be avoided in situations where comma is given a special meaning, for example in actual arguments to function calls and lists of initializers.

Declarations

Declarations are used within function definitions to specify the interpretation which Cpi gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form declaration: decl-specifiers declarator-listopt ; The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of at most one type-specifier and at most one storage class specifier.

```
decl-specifiers:
```

```
type-specifier

sc-specifier

type-specifier sc-specifier

sc-specifier type-specifier
```

TYPE SPECIFIERS

The type-specifiers are

type-specifier:

```
int

char

struct { type-decl-list }

struct identifier { type-decl-list }

struct identifier
```

if the type-specifier is missing from a declaration, it is generally taken to be int.

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 ( )

    declarator [ constant-expressionopt ]
```

```
( 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 for D a declarator, then the contained identifier has the type "pointer to . . .", where " . . . " is the type which the identifier would have had if the declarator had been simply D.

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.

A declarator may have the form

D[constant-expression] or D[ ]

In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is int. In the second the constant 1 is used. Such a declarator makes the contained identifier have type "array." If the unadorned declarator D would specify a non array of type ". . .", then the declarator "D[i]" yields a 1-dimensional array with rank i of objects of type ". . .". I

STRUCTURE DECLARATIONS

Recall that one of the forms for a structure specifier is

struct { type-decl-list }

The type-decl-list is a sequence of type declarations for the members of the structure:

```
type-decl-list:
```

type-declaration

```
type-declaration type-decl-list
```

A type declaration is just a declaration which does not mention a storage class (the storage class ''member of structure'' here being understood by context).

```
type-declaration:
```

```
type-specifier declarator-list ;
```

Within the structure, the objects declared have addresses which increase as their declarations are read left-to-right. Each component of a structure begins on an addressing boundary appropriate to its type. Another form of structure specifier is struct identifier `{ type-decl-list }` This form is the same as the one just discussed, except that the identifier is remembered as the structure tag of the structure specified by the list. A subsequent declaration may then be given using the structure tag but without the list, as in the third form of structure specifier:

```
struct identifier
```

Structure tags allow definition of self-referential structures; they also permit the long part of the declaration to be given once and used several times. It is however absurd to declare a structure which contains an instance of itself, as distinct from a pointer to an instance of itself. A simple example of a structure declaration where its use is illustrated more fully, is

```
struct tnode {
      char tword[20];
      int count;
      struct tnode *left;
      struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the following declaration makes sense:

```
struct tnode s, *sp;
```

which declares s to be a structure of the given sort and sp to be a pointer to a structure of the given sort. The names of structure members and structure tags may be the same as ordinary variables, since a distinction can be made by context. However, names of tags and members must be distinct. The same member name can appear in different structures only if the two members are of the same type and if their origin with respect to their structure is the same; thus separate structures can share a common initial segment.

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 substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the ‘‘else’’ ambiguity is resolved by connecting an else with the last encountered elseless if.

WHILE STATEMENT

The while statement has the form

```
while ( expression ) statement
```

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

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. If required, the expression is converted, as if by assignment, 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.

NULL STATEMENT

The null statement has the form

;

A null statement is useful to carry a label just before the ‘‘}’’ of a compound statement or to supply a null body to a looping statement such as while.

DEFINITIONS

A Cpi program consists of a sequence of Definitions. Definitions may be given for functions, for simple variables, and for arrays. They are used both to declare and to reserve storage for objects. A definition declares an identifier to have a specified type. The type-specifier may be empty, in which case the type is taken to be int.

3.1.1 FUNCTION DEFINITIONS

Function definitions have the form

```
function-definition:

     type-specifier-opt function-declarator function-body
```

A function declarator is similar to a declarator for a ‘‘function returning ...’’ except that it lists the formal parameters of the function being defined.

```
function-declarator:

     declarator ( parameter-listopt )

parameter-list:

     identifier

     identifier , parameter-list
```

The function-body has the form

```
function-body:
```

```
type-decl-list function-statement
```

The purpose of the type-decl-list is to give the types of the formal parameters. No other identifiers should be declared in this list, and formal parameters should be declared only here. The function-statement is just a compound statement which may have declarations at the start.

```
function-statement:

        { declaration-listopt statement-list }
```

Since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared ‘‘array of ...’’ are adjusted to read ‘‘pointer to ...’’. Finally, because neither structures nor functions can be passed to a function, it is useless to declare a formal parameter to be a structure or function (pointers to structures or functions are of course permitted). A free return statement is supplied at the end of each function definition, so running off the end causes control, but no value, to be returned to the caller.

Initializations

Arrays cannot be initialized during their declaration. Structures also cannot be initialized during their declaration. An example initialization for a structure and array are shown below.

```
struct tnode {
      char tword[20];
      int count;
};

tnode s;
s.count = 5;

int arr[3];
arr[0] = 1;
arr[1] = 2;
arr[2] = 3;
arr[3] = 4;

s.tword[0] = 'H';
s.tword[1] = 'e';
s.tword[2] = 'l';
s.tword[3] = 'l';
s.tword[4] = 'o';
```

Scope rules

A complete C program source text must be kept in a single file. This makes it possible for variables only to have a lexical scope. It is essentially the region of a program during which the identifier may be used without drawing "undefined identifier" diagnostics. It is an error to redeclare identifiers already declared in the current context.

Types revisited

This section summarizes the operations which can be performed on objects of certain types.

Structures

There are only two things that can be done with a structure: pick out one of its members (by means of the . or –> operators); or take its address (by unary &). Other operations, such as assigning from or to it or passing it as a parameter, draw an error message.

Functions

The only thing that can be done with a function is - call it.

Arrays, pointers, and subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator [ ] is interpreted in such a way that "E1[E2]" is identical to "*((E1) + (E2))". Because of the conversion rules which apply to +, if E1 is an array and E2 an integer, then E1[E2] refers to the E2-th member of E1. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

Examples.

These examples are intended to illustrate some typical C constructions as well as a serviceable style of writing C programs.

Inner product

This function returns the inner product of its array arguments.

```
int inner ( int v1[], int v2[], n ){
      int sum ;
      int i ;
      sum = 0 ;
      i=0;

      while (i<n){
```

```
        sum = sum + (v1 [ i ] * v2 [ i ]);
        i= i + 1;
    }
    return ( sum );
}
```

The following version is somewhat more efficient, but perhaps a little less clear. It uses the facts that parameter arrays are really pointers, and that all parameters are passed by value.

```
int inner ( int *v1, int  *v2, n )
{
    int sum ;
    sum = 0 ;
    while ( n ){
        *v1 = *v1 + 1;
        *v2 = *v2 + 1
        sum = sum + (*v1 * *v2);
        n = n - 1;
    }
    return ( sum ) ;
}
```

BINARYSEARCH.CPI

```
int binary_search(int array[], int start, int end, int element) {
    int mid;
    int temp;

    if (start > end){
    return -1;
    } else {
    mid = ((start + end)/2);
    temp = array[mid];
    if (temp == element) {
        return mid;
    } else if (temp > element) {
        return binary_search(array, start, mid - 1, element);
    } else {
        return binary_search(array, mid + 1, end, element);
    }
    }
}

int bin_search(int array[], int size, int element)
{
    return binary_search(array, 0, size - 1, element);
}

int main()
```

```
{
  int arr[10];
  int size;
  int target;
  int result;

  arr[0] = 1;
  arr[1] = 2;
  arr[2] = 4;
  arr[3] = 8;
  arr[4] = 16;
  arr[5] = 32;
  arr[6] = 64;
  arr[7] = 128;
  arr[8] = 256;
  arr[9] = 512;
  size = 10;
  target = 32;

  return bin_search(arr, size, target);
}
```

REFERENCES

Ritchie, Dennis M. "C reference manual." *Programming Languages*. Springer Berlin Heidelberg, 1983. 386-416.

1. Introduction
    - Include your language white paper.

4 PROJECT PLAN

Project planning began immediately after forming a team during the first 2 weeks of class. One of the biggest difficulties we encountered was actually deciding on which language to implement. The first month of the project was devoted to developing ideas and determining the feasibility of implementation. In the end, Professor Edwards helped us to narrow down the idea for developing a language for a specific architecture and we chose the Raspberry Pi due to its low cost, availability, popularity, and applicability to mobile hardware.

4.1 PROJECT MANAGEMENT

4.1.1 PLANNING

One of the first things we did was to assign a weekly meeting time to discuss project specific matters. We had a weekly meeting time each Monday and Wednesday after class to plan for the next features to add. In addition, we had regular meeting times on Wednesday with Professor Edwards to discuss current progress and problems.

4.1.2 *Specification*

We followed the ANSI C specification from Dennis Ritchie's C reference manual. We stripped down the features that we did not support and used that as the basis for our LRM. Although we originally planned to support global variables declarations, we were not able implement these in our final design and hence it was removed. Often, whenever there was a doubt of a specification, we looked to the gcc compiler on the Raspberry Pi. Unknown specifications for implementation would be coded for the Raspberry Pi and could then compiled with the –S option to produce assembly output. The code results and assembly output were then used to clear up any confusion about the specification and how the Cpi language should operate.

4.1.3 *Development*

We used Git as a distributed version control system to allow all members of the group to work independently. We used an iterative approach to software development and each group member worked on an individual feature at a time, merging the branch when all regression tests passed. In particular, we developed on several branches to implement features such as the type checking and refactoring offset calculation into our compilation phase. These changes were then merged back into the main branch once we achieved a stable commit. Finally, we made extensive use of the issue tracker feature on Github to discuss future feature additions and keep track of problems and implementation details. When issues such as bugs presented themselves, they were assigned to the appropriate team member to be handle.

4.1.4 *Testing*

Early on in the development process we largely implemented features without having tests to verify them. Tests were often painful to write and run due the generation of an ARM executable as final product and our use of x86 based architectures. While originally we had hoped to perform all tests on a QEMU emulation of the Raspberry Pi architecture, we encountered problems in transferring source files and executables to the QEMU emulator. Even when we were successful in transferring files, performance on the emulator was sometimes slow and unresponsive. Our testing setup was vastly improved when we transferred testing to a dedicated Raspberry Pi server. An OCaml build environment was setup and our git repository was cloned onto the Pi. This enabled teammates to directly SSH onto the Pi and develop and run tests. Tests were written for existing features and from that point forwards, we followed a test driven approach, building tests for the compiler before and while implementing features.

4.2 Style Guide

Since this was our first time coding a project in OCaml there was no set style guidelines at the beginning of the project. However, as time progressed there quickly became the need to write code to a common style guide to maintain readability. In particular, the following rules were generally applied:

1.) Maximum length of a single line must not exceed 80 characters
2.) Each code block following a Let .. in statement must be indented.

3.) Underscore casing for all variable and function names
4.) One statement per line
5.) Break up complex function into smaller functions as much as possible.

4.3 PROJECT TIMELINE

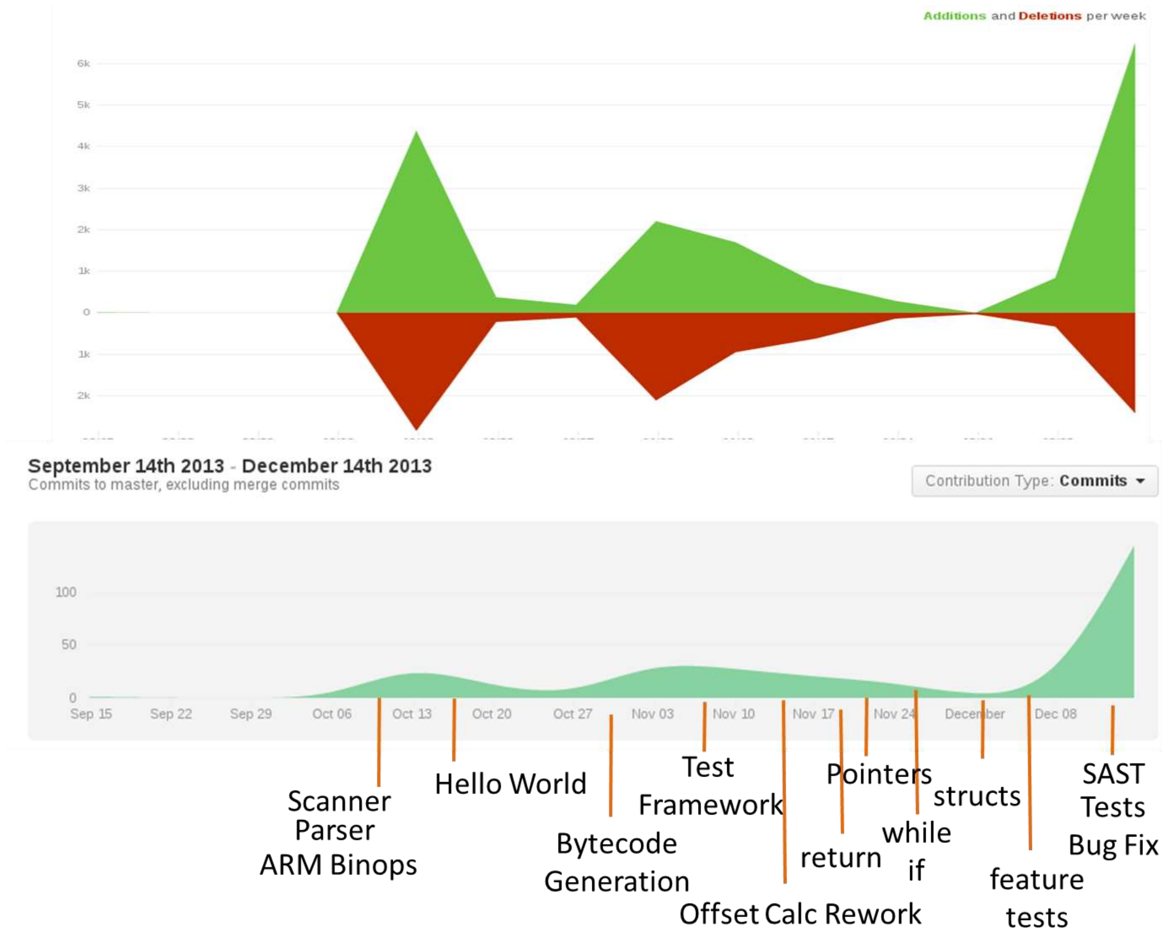


FIGURE 1. PROJECT TIMELINE BY CODE ADDITION AND COMMITS

Figure 1 shows the project timeline for the Cpi compiler. The top graph shows number of lines of code additions in green and code deletions in red. The bottom graph shows the total number of commits over the course of the project (352 commits in all). Work began in earnest after we submitted the second homework assignment on October 14th. Work continued at steady pace with decreases during Thanksgiving and midterm weeks. A large amount of commits is observed close to the deadline to submission after the addition of a large amount of tests and bug fixes. A chart with the dates of major accomplishments is shown in Table 2.

Date	Accomplishment
10/9/13	Scanner, Parser, and generation of ARM code for simple binary operations
10/16/13	Hello World; strings and printf support for simple ARM code generation
10/28/13	Simple compilation of binary operation to bytecode and then to ARM assembly
11/6/13	New test framework base on Raspberry Pi Server
11/13/13	Refactor of bytecode generation into compilation stages
11/25/13	While, if, and pointer features.
12/8/13	Type checking, type checking tests, and numerous bugs fixes.

TABLE 2. MAJOR ACCOMPLISHMENTS AND DATES

4.4 SOFTWARE DEVELOPMENT ENVIRONMENT

The following tools and software packages were used to develop Cpi:

1.) Linux development environment on x86 machines
2.) Raspberry Pi running the 2013-09-10 image of Raspbian
3.) Qemu emulation environment with ARM1176JZF-S libraries
4.) OCaml
5.) VIM text editor for IDE
6.) GCC compiler and linker on Raspberry Pi
7.) Git verion source control

4.5 ROLES AND RESPONSIBILITIES

Edward Garcia - Type Checking, Test Case Generation, External functions

Niket Kandya - Scanner/Parser, Scalar Types and Functions, Design

Naveen Revanna – System Architect, Bytecode Generation

Sean Yeh - Test suite, Example programs, Bug Fixes,

5 Architectural Design

5.1 Overview

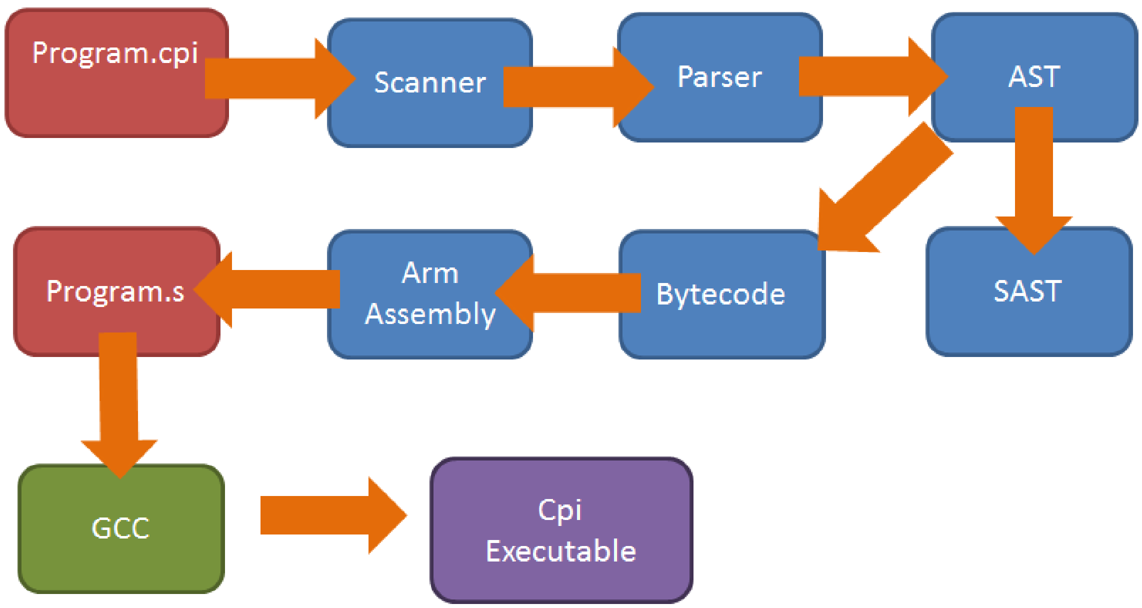


FIGURE 2. OVERVIEW OF SYSTEM ARCHIETECTURE

The architectural design of Cpi is shown in Figure 2. Input and out files are shown in red and components of the compiler are shown in blue. Overall, Cpi follows a traditional compiler design with a lexical scanner 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 ARM assembly code generation from an intermediate bytecode representation.

5.2 SCANNING, PARSING, AND AST

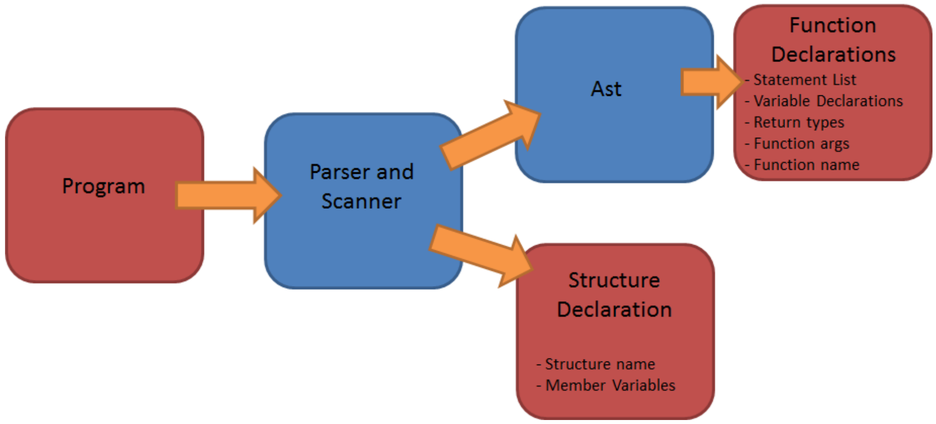


FIGURE 3. SCANNING, PARSING AND AST GENERATION

Figure 3 shows the entry components to the Cpi compiler. 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 Cpi 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 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. Table 3 gives the datatypes for the AST generation.

Expressions	Description
Literal (value)	A constant number to be used in an expression.
String (value)	A constant string of type [Ptr;Char]
Addrof (expression)	Operation to return the address of the result of expression
Negof (expression)	Operation to apply take the negative of the result of expression
ConstCh (char_val)	A single constant character of type [Char]
Id (name)	Operation to return the information of variable with name
MultiId(struct_id_expression, resolve_operator, member_id_expression)	Structure dereferencing operation. struct_id_expression can be an array, pointer or constant.
Pointer (expression)	Expression to return the value in the address calculated by expression
Array (id_expression, index_expression)	Operation to get the variable information of id_expression and apply an offset of the result of index _expression
Binop (expression1, operation, expression2)	Applies operation to expression1 and expression2
Assign (expression1, expression2)	Assigns the value of the result of expression2 to expression 1.
Call (function_name, parameter_list)	An expression to branch to the function with function_name and pass the expression list as parameter_list
Null	A void pointer constant with value of 0
Noexpr	No operation to perform

TABLE 3. EXPRESSIONS USED IN AST CREATION

Statements
Block (statement list)
Expr (expression)
Return (expression)
If(expression, true_statement, false_statement)
For (asn_expr, check_expr, incr_expr)
While (condition_expr, statement_block)

TABLE 4. STATEMENTS USED IN AST CREATION

5.3 SAST CREATION

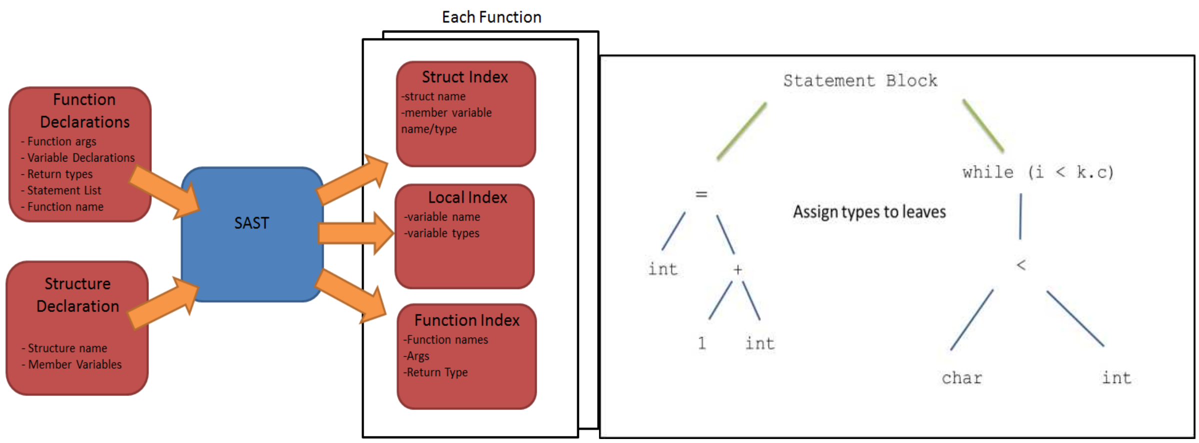


FIGURE 4. SAST CREATION FROM INDEXES

Figure 4 show the creation of the SAST from the outputs of the Parser and AST components. 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 table of these checks and error resulting from mismatched types is shown in Table 5.

Checks	Error
While conditions	conditional expression must be int or char
If conditions	conditional expression must be int or char
Variable assignments	Left hand and right hand must match
Variable assignments	Left hand cannot be an array or address (i.e. &a = 4)
Variable declarations	Variable cannot be declared twice
Binary Operations	Left hand and right hand must match
Pointer arithmetic	Limited to addition, subtraction, and comparison operators (<, >, ==, !=)
Function arguments	Function call argument types must match function declaration types
Function declarations	Functions cannot be declared twice
Function return	Function return must match return type in function declaration
Unary (-)	Cannot have a negative pointer, struct or address

TABLE 5. TYPE CHECKING RULES

While Cpi shares many of the type checking rules as C, one divergent aspects is that warnings raised by C are raised as failures in Cpi. Cpi is statically typed and allows operations to be performed with interchangeable int and char types.

5.4 BYTECODE GENERATION/COMPILER

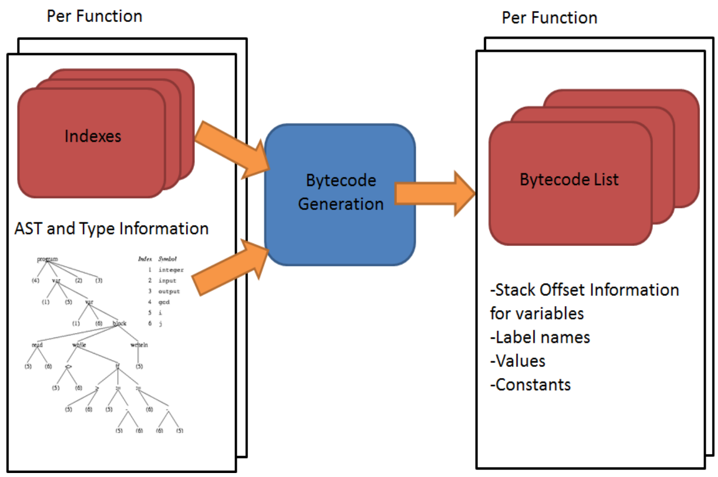


TABLE 6. BYTECODE GENERATION

Using the same method of assigning types to all nodes of the tree, Figure 6 shows the bytecode generation component creating a list of bytecode statements for each function in the program. The list of bytecode statements is shown in Table 7.

The main funciton of the compiler is to convert the Ast tree into a flattened list of bytecodes. The advantage of the flattened bytecode list is that can be used by the code generator to generate code looking at individual elements and there is no need for any information of its predecessor or successors unlike in the Ast tree.

Atomics
Lit (Literal_value)
Cchar (char)
Sstr(string_value, label)
Lvar(offset,size)
Gvar (name,size)
Pntr (addr,size)
Addr (atom)
Neg (atom)
Debug (debug_string)

TABLE 7. ATOMIC USED IN BYTECODE GENERATION

Bytecode Statements
Atom(atom)
VarArr (atom, atom)
Rval (atom)
BinEval (atom, atom, operation, atom)
BinRes (cpitypes list)
Assgmt (left_hand, right)hand)
Fcall (func_name, atom list, atom)
Branch (label)
Predicate (var_to_check, jump_on_what? , label)
Label (label_name)

TABLE 8. BYTECODE STATEMENTS

Lets take an example to understand the conversion of the Ast tree to bytecode.

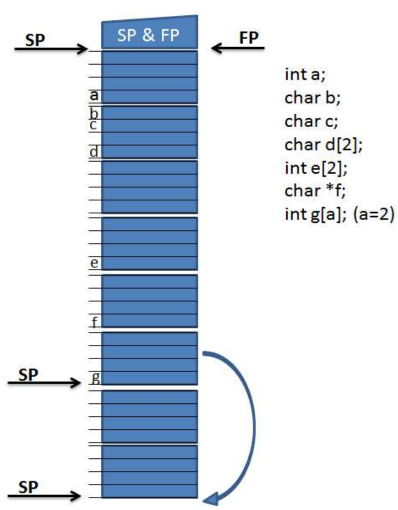
Cpi code snippet: arr[a + b + 2]

Ast tree: Array (Id (a), Binop(Binop(a, Add, b), Add, Lit(2)))

Bytecode: [BinEval(t1,a,+,b); BinEval(t2,t1,+,2); BinEval(t3,t2,*,4); BinEval(t4,Addr(arr),+,t3); Pntr(t4)]

Note: the variables t1,a etc in the Bytecode will have a representaiton of the form Lvar(offset,size).

As can be seen above, the ast tree is being converted to flattened bytecode which the code generator will use to convert appropriate assembly code. Figure 9 shows how Cpi generates the offsets calculations for each bytecode using the type information of each expression. Using a stack in descending order, each local variable declared in a function is allocated on the stack using 4 byte alignments. Structures are allocated on the stack similarly but in reverse order with the base address at the lowest address and member variable allocated above it.



Variable Symbol Table

varname	type	offset
a	[Int]	4
b	[Char]	5
c	[Char]	6
d	[Arr(2);Char]	8
e	[Arr(2);Int]	16
f	[Ptr;Char]	20
g	[Ptr;Int]	24

TABLE 9. OFFSET CALCULATION

5.5 ARM Assembly Information

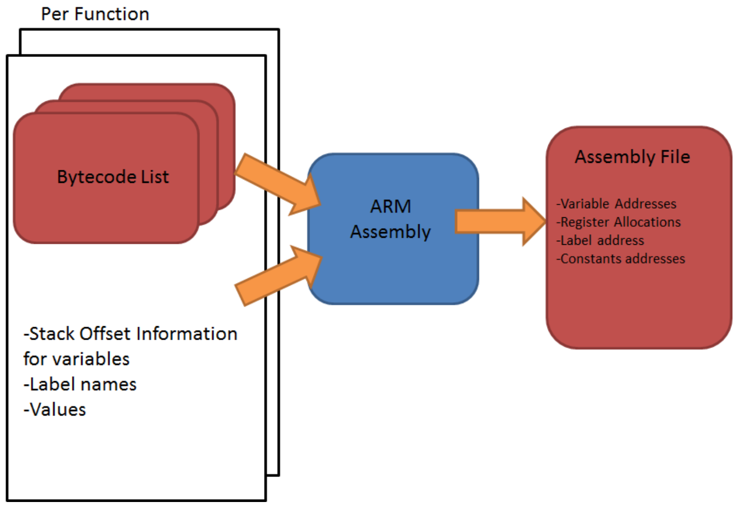


TABLE 10. ARM CODE GENERATION FROM BYTECODE

Figure 10 shows the final step in the compilation process, the creation a single ARM assembly file from a list of bytecode statements. During this final step, registers are assigned, addresses are computed for variables and references/constants are allocated in memory. The generated ARM assembly is completely unoptimized.

6 Test Plan

6.1 Automation Script

The testing script is a BASH script that reads every test file in the testing directory, compiles it with both gcc and cpi, and compare the outputs from printf and exit codes. It will also display whether or not there was a gcc compile error, cpi compile error, or assembler error. As with much of our project, we take advantage of make tasks. The command `make test` will run the test suite on the pi, and `make test_edpi` will ssh into the pi, pull in the latest changes, and run `make test` on the pi.

6.2 Test Case Selection

We tried to make tests for every feature of the language before the implementing it and were fairly successful in making an exhaustive test suite (of over 150 tests). Using a shell script that automated the test suite running on an actual raspberry pi, running the tests was very simple, and creating tests simply involved adding a file to the tests directory.

We have two kinds of tests: features tests and type checking test. For each feature in Cpi, we implement several feature tests. For each type checking error that Cpi gives, we provide several type checking tests. In the feature tests, all the files are *.cpi files (with the exception of the tests for scanf that are ignored during automatic testing, which are given a .c extension) that are expected to pass. The typechecking tests are located in the different directory, but the testing script is almost the same. In these tests, we specify whether tests should pass or fail by giving them a .pass or .fail file extension, and are expected to pass or fail according to their file extension.

6.3 FEATURE TESTS SUITE:

arrayasargument.cpi
arrayvarsize.cpi
arrfunc.cpi
arrvarindex1.cpi
arrvarindex2.cpi
arrvarindex.cpi
assign1.cpi
assign2.cpi
assign3.cpi
assign4.cpi
assign5.cpi
binsearch.cpi
char1.cpi
char2.cpi
char3.cpi
charptr2.cpi
charptr3.cpi
charptr4.cpi
charptr.cpi
charptrmod.cpi
charr2.cpi
charr.cpi
commentblock.cpi
commentnested.cpi
commentslash.cpi
div1.cpi
div2.cpi
elseif.cpi
expr1.cpi
expr2.cpi
expr3.cpi
expr4.cpi
expr5.cpi
for2.cpi
for.cpi
functions.cpi
gcd.cpi
if2.cpi
if3.cpi
if4.cpi
if5.cpi
if6.cpi
if_conditionals2.cpi
if_conditionals.cpi
if.cpi
inner2.cpi
inner.cpi
intarr1.cpi
intarr2.cpi
intarr.cpi
intarrptr.cpi
intptr.cpi
intptrmod2.cpi
intptrmod.cpi
linearsearch_negative.cpi
linearsearch_positive.cpi
logical_and2.cpi
logical_and.cpi
logical_or.cpi
malloc.cpi
multirecursion.cpi
neg1.cpi
neg2.cpi
neg.cpi
pointers1.cpi
pointers2.cpi
pointers3.cpi
print2.cpi
print3.cpi
recursionAddition.cpi
scan1.c
scan2.c
scan3.c
selectionsort.cpi
struct1.cpi
struct2.cpi
structarray.cpi
structbasic.cpi
structfunc2.cpi
structfunc.cpi
structptr1.cpi
structptr2.cpi
structptr3.cpi
structptr4.cpi
structptrarg1.cpi
structptrarg2.cpi
structptr.cpi
structtest1.cpi
structtest2.cpi
structtest3.cpi
structtest4.cpi
test1.cpi
test2.cpi
test3.cpi
varname.cpi
while1.cpi
while2.cpi
while3.cpi
while4.cpi
while5.cpi

6.4 TYPECHECKING TEST SUITE

charptr.pass
functions.pass
if.pass

intarr.pass
intarrptr.pass
intptrmod.pass
intptr.pass
struct5.pass
struct6.pass
struct7.pass
struct.pass
while.pass
assign1.fail
assign2.fail
assign3.fail
assign4.fail
assign5.fail
assign6.fail
charptr1.fail
charptr2.fail
charptr3.fail
charptr4.fail
charptr5.fail
charptr6.fail
charptr7.fail
func1.fail
functions1.fail
functions2.fail
functions3.fail
functions4.fail
functions5.fail
functions6.fail
functions7.fail
if1.fail
if2.fail
if3.fail
if4.fail
intarr1.fail
intarr2.fail
intarr3.fail
intarr4.fail
intarr5.fail
intarr6.fail
intptr1.fail
intptr2.fail
intptr3.fail
intptr4.fail
intptr5.fail
intptr6.fail
intptrmod1.fail
intptrmod2.fail
neg1.fail
neg2.fail
neg3.fail
struct1.fail
struct2.fail
struct3.fail
struct4.fail
struct5.fail
struct6.fail
while1.fail
while2.fail
while3.fail
while4.fail

6.5 DEMO PROGRAMS

6.5.1 *MAKEFILE*

```
bf:
	../cpi bf.cpi -o bf.out --binary

bf_test: bf
	./runbf.sh "++++++++++[>+++++++>++++++++++>+++>+<<<<-
]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.>."


tictactoe :
	../cpi tictactoe.cpi -o tictactoe.out --binary

ll:
	../cpi linked_list.cpi -o linked_list.out --binary

.PHONY : clean
clean :
	rm -f *.out
```

6.5.2 *RUNBF.SH*

```
#!/bin/zsh
source=$1
```

```
temp=`echo $source | wc -c`
len=`echo "$temp - 1" | bc`

{echo $len; echo $source} | ./bf.out
```

6.5.3 *BF.CPI*

```
/* Special thanks to: https://github.com/mig-hub/yabi */


int do_command(char dh[], char command, char source[], int* dh_index, int index)
{
    char c;
    int pos;
    char *p;
    int tempbreak;
    int loopc;

    /* printf("index:%d\n, command:%c\n",index,command); */
    /* printf("cell[0]: %d, cell[1]: %d\n",dh[0], dh[1]); */
    /* printf("dh_index:%d\n",*dh_index); */

    p = &dh[*dh_index];

    if (command == '>'){
        *dh_index = *dh_index + 1;
        return index;
    }
    if (command == '<'){
        *dh_index = *dh_index - 1;
        return index;
    }
    if (command == '+'){
        *p =  *p + 1;
        return index;
    }
    if (command == '-'){
        *p = *p - 1;
        return index;
    }
    if (command == '.'){
        printf("%c",*p);
        return index;
    }
    if (command == ','){
        scanf(" %c", p);
        return index;
```

```
    }
    if (command == '['){
        pos = index;
        if ((*p) == 0) {
            loopc = 0;
            tempbreak = 0;
            while(tempbreak == 0)
            {
                index = index + 1;
                c = source[index];

                if (loopc == 0){
                    if (c == ']'){
                        tempbreak = 1;
                    }
                }

                if (tempbreak == 0){
                    if (c == ']'){
                        loopc = loopc - 1;
                    }

                    if (c == '['){
                        loopc = loopc + 1;
                    }
                }
            }
        } else {
            while((*p) != 0) {
                index = pos;

                index = index + 1;
                c = source[index];

                while( c!=']') {
                    index = do_command(dh, c, source, dh_index, index);

                    index = index + 1;
                    c = source[index];
                }

            }
        }

    }
    return index;
}
```

```
int main() {
    char command;
    int len;
    char source[500];
    char data_highway[100];
    char *p;
    int index;
    int newindex;
    int x;
    int *dh_index;

    x = 0;
    dh_index = &x;

    scanf("%d", &len);
    scanf("%s", source);

    printf("len: %d, source: %s\n", len, source);

    index = 0;
    p = &data_highway[0];
    while(index < len) {
        command = source[index];

        newindex = do_command(data_highway, command, source, dh_index, index);
        index = newindex + 1;
    }
    printf("\n");

    return 0;
}
```

6.5.4 BF.S

```
.data
.LC0:
        .asciz   "%d"
.LC1:
        .asciz   "%s"
.LC2:
        .asciz   "len: %d, source: %s\n"
.LC3:
        .asciz   "\n"

.text
.global main
main:
        stmfd sp!, {fp, lr}
        add fp, sp,#4
        sub sp, sp,#1264
        ldr r0, =0
        str r0, [fp,#-628]
        sub r0, fp,#628
        str r0, [fp,#-632]

        ldr r0, =.LC0
        sub r1, fp,#12
        bl  scanf
        str r0, [fp,#-636]

        ldr r0, =.LC1
        sub r1, fp,#512
        bl  scanf
        str r0, [fp,#-640]

        ldr r0, =.LC2
        ldr r1, [fp,#-12]
        sub r2, fp,#512
```

```
        bl  printf
        str r0, [fp,#-
644]
        ldr r0, =0
        str r0, [fp,#-
620]
        ldr r0, =1
        ldr r1, =0
        muls r3, r0, r1
        str r3, [fp,#-
648]
        sub r0, fp,#612
        ldr r1, [fp,#-
648]
        adds r3, r0, r1
        str r3, [fp,#-
748]
        ldr r0, [fp,#-
748]
        str r0, [fp,#-
616]
        b loop1_end
loop1_start:
        ldr r0, =1
        ldr r1, [fp,#-
620]
        muls r3, r0, r1
        str r3, [fp,#-
752]
        sub r0, fp,#512
        ldr r1, [fp,#-
752]
        adds r3, r0, r1
        str r3, [fp,#-
1252]
        ldr r4, [fp,#-
1252]
        ldrb r0, [r4,#0]
        strb r0, [fp,#-5]
        sub r0, fp,#612
        ldrb r1, [fp,#-5]
        sub r2, fp,#512
        ldr r3, [fp,#-
632]
        ldr r4, [fp,#-
620]
        bl  do_command
        str r0, [fp,#-
1256]
        ldr r0, [fp,#-
1256]
        str r0, [fp,#-
624]
        ldr r0, [fp,#-
624]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-
1260]
        ldr r0, [fp,#-
1260]
        str r0, [fp,#-
620]
loop1_end:
        ldr r0, [fp,#-
620]
        ldr r1, [fp,#-12]
        cmp r0, r1
        movlt r3,#1
        movge r3,#0
        uxtb r3,r3
        str r3, [fp,#-
1264]
        ldr r0, [fp,#-
1264]
        cmp r0,#0
        bne loop1_start
        ldr r0, =.LC3
        bl  printf
        str r0, [fp,#-
1268]
        ldr r0, =0
        b main_exit
main_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}

.data
.LC4:
        .asciz   "%c"
.LC5:
        .asciz   " %c"
.text
.global do_command
do_command:
        stmfd sp!, {fp,
lr}
        add fp, sp,#4
        sub sp, sp,#184
        str r0, [fp,#-28]
        strb r1, [fp,#-
29]
        str r2, [fp,#-36]
        str r3, [fp,#-40]
        str r4, [fp,#-44]
        ldr r0, =1
        ldr r4, [fp,#-40]
        ldr r1, [r4,#0]
        muls r3, r0, r1
        str r3, [fp,#-48]
        ldr r0, [fp,#-28]
        ldr r1, [fp,#-48]
        adds r3, r0, r1
        str r3, [fp,#-52]
        ldr r0, [fp,#-52]
        str r0, [fp,#-16]
        ldrb r0, [fp,#-
29]
        ldrb r1, =62
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
53]
        ldrb r0, [fp,#-
53]
        cmp r0,#0
        beq end1
        ldr r4, [fp,#-40]
        ldr r0, [r4,#0]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-60]
        ldr r0, [fp,#-60]
        ldr r4, [fp,#-40]
        str r0, [r4,#0]
        ldr r0, [fp,#-44]
        b do_command_exit
```

```
end1:
        ldrb r0, [fp,#-
29]
        ldrb r1, =60
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
61]
        ldrb r0, [fp,#-
61]
        cmp r0,#0
        beq end2
        ldr r4, [fp,#-40]
        ldr r0, [r4,#0]
        ldr r1, =1
        subs r3, r0, r1
        str r3, [fp,#-68]
        ldr r0, [fp,#-68]
        ldr r4, [fp,#-40]
        str r0, [r4,#0]
        ldr r0, [fp,#-44]
        b do_command_exit
end2:
        ldrb r0, [fp,#-
29]
        ldrb r1, =43
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
69]
        ldrb r0, [fp,#-
69]
        cmp r0,#0
        beq end3
        ldr r4, [fp,#-16]
        ldrb r0, [r4,#0]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-76]
        ldr r0, [fp,#-76]
        ldr r4, [fp,#-16]
        strb r0, [r4,#0]
        ldr r0, [fp,#-44]
        b do_command_exit
end3:
        ldrb r0, [fp,#-
29]
        ldrb r1, =45
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
77]
        ldrb r0, [fp,#-
77]
        cmp r0,#0
        beq end4
        ldr r4, [fp,#-16]
        ldrb r0, [r4,#0]
        ldr r1, =1
        subs r3, r0, r1
        str r3, [fp,#-84]
        ldr r0, [fp,#-84]
        ldr r4, [fp,#-16]
        strb r0, [r4,#0]
        ldr r0, [fp,#-44]
        b do_command_exit
end4:
        ldrb r0, [fp,#-
29]
        ldrb r1, =46
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
85]
        ldrb r0, [fp,#-
85]
        cmp r0,#0
        beq end5
        ldr r0, =.LC4
        ldr r4, [fp,#-16]
        ldrb r1, [r4,#0]

        bl  printf
        str r0, [fp,#-92]
        ldr r0, [fp,#-44]
        b do_command_exit
end5:
        ldrb r0, [fp,#-
29]
        ldrb r1, =44
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
93]
        ldrb r0, [fp,#-
93]
        cmp r0,#0
        beq end6
        ldr r0, =.LC5
        ldr r1, [fp,#-16]

        bl  scanf
        str r0, [fp,#-
100]
        ldr r0, [fp,#-44]
        b do_command_exit
end6:
        ldrb r0, [fp,#-
29]
        ldrb r1, =91
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
101]
        ldrb r0, [fp,#-
101]
        cmp r0,#0
        beq end13
        ldr r0, [fp,#-44]
        str r0, [fp,#-12]
        ldr r4, [fp,#-16]
        ldrb r0, [r4,#0]
        ldr r1, =0
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-
108]
        ldr r0, [fp,#-
108]
```

```
        cmp r0,#0
        beq else12
        ldr r0, =0
        str r0, [fp,#-24]
        ldr r0, =0
        str r0, [fp,#-20]
        b loop2_end
loop2_start:
        ldr r0, [fp,#-44]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-
112]
        ldr r0, [fp,#-
112]
        str r0, [fp,#-44]
        ldr r0, =1
        ldr r1, [fp,#-44]
        muls r3, r0, r1
        str r3, [fp,#-
116]
        ldr r0, [fp,#-36]
        ldr r1, [fp,#-
116]
        adds r3, r0, r1
        str r3, [fp,#-
120]
        ldr r4, [fp,#-
120]
        ldrb r0, [r4,#0]
        strb r0, [fp,#-5]
        ldr r0, [fp,#-24]
        ldr r1, =0
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-
124]
        ldr r0, [fp,#-
124]
        cmp r0,#0
        beq end8
        ldrb r0, [fp,#-5]
        ldrb r1, =93
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
125]
        ldrb r0, [fp,#-
125]
        cmp r0,#0
        beq end7
        ldr r0, =1
        str r0, [fp,#-20]
end7:
end8:
        ldr r0, [fp,#-20]
        ldr r1, =0
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-
132]
        ldr r0, [fp,#-
132]
        cmp r0,#0
        beq end11
        ldrb r0, [fp,#-5]
        ldrb r1, =93
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
133]
        ldrb r0, [fp,#-
133]
        cmp r0,#0
        beq end9
        ldr r0, [fp,#-24]
        ldr r1, =1
        subs r3, r0, r1
        str r3, [fp,#-
140]
        ldr r0, [fp,#-
140]
        str r0, [fp,#-24]
end9:
        ldrb r0, [fp,#-5]
        ldrb r1, =91
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
141]
        ldrb r0, [fp,#-
141]
        cmp r0,#0
        beq end10
        ldr r0, [fp,#-24]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-
148]
        ldr r0, [fp,#-
148]
        str r0, [fp,#-24]
end10:
end11:
loop2_end:
        ldr r0, [fp,#-20]
        ldr r1, =0
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-
152]
        ldr r0, [fp,#-
152]
        cmp r0,#0
        bne loop2_start
        b end12
else12:
        b loop4_end
loop4_start:
        ldr r0, [fp,#-12]
        str r0, [fp,#-44]
        ldr r0, [fp,#-44]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-
156]
        ldr r0, [fp,#-
156]
        str r0, [fp,#-44]
        ldr r0, =1
        ldr r1, [fp,#-44]
        muls r3, r0, r1
```

```
        str r3, [fp,#-
160]
        ldr r0, [fp,#-36]
        ldr r1, [fp,#-
160]
        adds r3, r0, r1
        str r3, [fp,#-
164]
        ldr r4, [fp,#-
164]
        ldrb r0, [r4,#0]
        strb r0, [fp,#-5]
        b loop3_end
loop3_start:
        ldr r0, [fp,#-28]
        ldrb r1, [fp,#-5]
        ldr r2, [fp,#-36]
        ldr r3, [fp,#-40]
        ldr r4, [fp,#-44]

        bl  do_command
        str r0, [fp,#-
168]
        ldr r0, [fp,#-
168]
        str r0, [fp,#-44]
        ldr r0, [fp,#-44]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-
172]
        ldr r0, [fp,#-
172]
        str r0, [fp,#-44]
        ldr r0, =1
        ldr r1, [fp,#-44]
        muls r3, r0, r1
        str r3, [fp,#-
176]
        ldr r0, [fp,#-36]
        ldr r1, [fp,#-
176]
        adds r3, r0, r1
        str r3, [fp,#-
180]
        ldr r4, [fp,#-
180]
        ldrb r0, [r4,#0]
        strb r0, [fp,#-5]
loop3_end:
        ldrb r0, [fp,#-5]
        ldrb r1, =93
        cmp r0, r1
        moveq r3,#0
        movne r3,#1
        uxtb r3,r3
        strb r3, [fp,#-
181]
        ldrb r0, [fp,#-
181]
        cmp r0,#0
        bne loop3_start
loop4_end:
        ldr r4, [fp,#-16]
        ldrb r0, [r4,#0]
        ldr r1, =0
        cmp r0, r1
        moveq r3,#0
        movne r3,#1
        uxtb r3,r3
        str r3, [fp,#-
188]
        ldr r0, [fp,#-
188]
        cmp r0,#0
        bne loop4_start
end12:
end13:
        ldr r0, [fp,#-44]
        b do_command_exit
do_command_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}
```

6.5.5 *LINKED_LIST.CPI*

```
#include<stdio.h>
#include<stdlib.h>


struct node
{
     struct node *previous;
     int data;
     struct node *next;
};
```

```
void insert_beginning(int value, struct node **head, struct node **last)
{
     struct node *var;
     struct node *temp;
     struct node *temp2;
          var=malloc(24);
     var->data = value;
     if(*head==NULL)
     {
            printf("Adding to Empty List\n");
         var->previous=NULL;
         var->next=NULL;
         *head = var;
         *last = *head;
     }
     else
     {
          printf("Adding to List\n");
         temp = var;
         temp->previous=NULL;
         temp->next = *head;
             (*head)->previous =  temp;
         *head = temp;
     }
}

int delete_from_end(struct node **head, struct node **last)
{
      struct node *temp;
      temp=*head;
      if(temp==NULL)
      {
         printf("Cannot Delete: ");
         return 0;
      }

      temp = *last;

      if(temp->previous == NULL)
      {
           printf("\nData deleted from list is %d \n",(*last)->data);
           free(temp);
           *head=NULL;
           *last=NULL;
           return 0;
      }

      printf("\nData deleted from list is %d \n",(*last)->data);
```

```
      *last = temp->previous;
      (*last)->next=NULL;
      free(temp);
      return 0;
}

void display(struct node **head, struct node **last)
{
     struct node *temp;
     temp=*head;
     if(temp==NULL)
      {
         printf("List is Empty!");
      }
     while(temp!=NULL)
     {
          printf("-> %d ",temp->data);
          temp=temp->next;
     }
}

int main()
{
    int value;
    int i;
    int loc;
    struct node *head;
    struct node *last;

    head = NULL;

    printf("Select the choice of operation on link list");
    printf("\n1.) insert at beginning\n");
    printf("2.) delete from end\n");
    printf("3.) display list\n");
    printf("4.) Exit\n");
    while(1)
    {
        printf("\n\nenter the choice of operation you want to do ");
        scanf("%d",&i);

        if (i == 1){
            printf("enter the value you want to insert in node ");
            scanf("%d",&value);
            insert_beginning(value, &head, &last);
            display(&head, &last);
        } else {if (i == 2){
            delete_from_end(&head, &last);
```

```
            display(&head, &last);
        } else {if (i == 3){
            display(&head, &last);
        } else {if (i == 4){
            return 0;
        } else {
            return 0;
        }}}}
    }
}
```

6.5.6 LINKED_LIST.S

```
.data
.LC0:
        .asciz  "Select the choice of operation on link list"
.LC1:
        .asciz  "\n1.) insert at beginning\n"
.LC2:
        .asciz  "2.) delete from end\n"
.LC3:
        .asciz  "3.) display list\n"
.LC4:
        .asciz  "4.) Exit\n"
.LC5:
        .asciz  "\n\nenter the choice of operation you want to do "
.LC6:
        .asciz  "%d"
.LC7:
        .asciz  "enter the value you want to insert in node "
.LC8:
        .asciz  "%d"

.text
.global main
main:
        stmfd sp!, {fp, lr}
        add fp, sp,#4
        sub sp, sp,#76
        ldr r0, =0
        str r0, [fp,#-20]
        ldr r0, =.LC0

        bl  printf
        str r0, [fp,#-28]
        ldr r0, =.LC1

        bl  printf
        str r0, [fp,#-32]
        ldr r0, =.LC2

        bl  printf
        str r0, [fp,#-36]
        ldr r0, =.LC3

        bl  printf
        str r0, [fp,#-40]
        ldr r0, =.LC4

        bl  printf
        str r0, [fp,#-44]
        b loop1_end
loop1_start:
        ldr r0, =.LC5

        bl  printf
        str r0, [fp,#-48]
        ldr r0, =.LC6
        sub r1, fp,#12
        bl  scanf
        str r0, [fp,#-52]
        ldr r0, [fp,#-12]
        ldr r1, =1
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-56]
        ldr r0, [fp,#-56]
        cmp r0,#0
        beq else4
        ldr r0, =.LC7

        bl  printf
        str r0, [fp,#-60]
        ldr r0, =.LC8
        sub r1, fp,#8

        bl  scanf
        str r0, [fp,#-64]
        ldr r0, [fp,#-8]
        sub r1, fp,#20
        sub r2, fp,#24

        bl  insert_beginning
        sub r0, fp,#20
        sub r1, fp,#24

        bl  display
        b end4
else4:
```

```
        ldr r0, [fp,#-12]
        ldr r1, =2
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-68]
        ldr r0, [fp,#-68]
        cmp r0,#0
        beq else3
        sub r0, fp,#20
        sub r1, fp,#24

        bl
delete_from_end
        str r0, [fp,#-72]
        sub r0, fp,#20
        sub r1, fp,#24

        bl  display
        b end3
else3:
        ldr r0, [fp,#-12]
        ldr r1, =3
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-76]
        ldr r0, [fp,#-76]
        cmp r0,#0
        beq else2
        sub r0, fp,#20
        sub r1, fp,#24

        bl  display
        b end2
else2:
        ldr r0, [fp,#-12]
        ldr r1, =4
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-80]
        ldr r0, [fp,#-80]
        cmp r0,#0
        beq else1
        ldr r0, =0
        b main_exit
        b end1
else1:
        ldr r0, =0
        b main_exit
end1:
end2:
end3:
end4:
loop1_end:
        ldr r0, =1
        cmp r0,#0
        bne loop1_start
main_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}

.data
.LC9:
        .asciz   "List is
Empty!"
.LC10:
        .asciz   "-> %d "

.text
.global display
display:
        stmfd sp!, {fp,
lr}
        add fp, sp,#4
        sub sp, sp,#44
        str r0, [fp,#-12]
        str r1, [fp,#-16]
        ldr r4, [fp,#-12]
        ldr r0, [r4,#0]
        str r0, [fp,#-8]
        ldr r0, =12
        ldr r1, =0
        muls r3, r0, r1
        str r3, [fp,#-24]
        ldr r0, [fp,#-8]
        ldr r1, [fp,#-24]
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-20]
        ldr r0, [fp,#-20]
        cmp r0,#0
        beq end5
        ldr r0, =.LC9

        bl  printf
        str r0, [fp,#-28]
end5:
        b loop2_end
loop2_start:
        ldr r0, [fp,#-8]
        ldr r1, =4
        adds r3, r0, r1
        str r3, [fp,#-32]
        ldr r0, =.LC10
        ldr r4, [fp,#-32]
        ldr r1, [r4,#0]

        bl  printf
        str r0, [fp,#-36]
        ldr r0, [fp,#-8]
        ldr r1, =8
        adds r3, r0, r1
        str r3, [fp,#-40]
        ldr r4, [fp,#-40]
        ldr r0, [r4,#0]
        str r0, [fp,#-8]
loop2_end:
        ldr r0, =12
        ldr r1, =0
        muls r3, r0, r1
        str r3, [fp,#-48]
        ldr r0, [fp,#-8]
        ldr r1, [fp,#-48]
        cmp r0, r1
        moveq r3,#0
        movne r3,#1
        uxtb r3,r3
        str r3, [fp,#-44]
        ldr r0, [fp,#-44]
        cmp r0,#0
        bne loop2_start
display_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}
```

```
.data
.LC11:
        .asciz   "Cannot 
Delete: "
.LC12:
        .asciz   "\nData 
deleted from list is %d
\n"
.LC13:
        .asciz   "\nData 
deleted from list is %d
\n"

.text
.global delete_from_end
delete_from_end:
        stmfd sp!, {fp, 
lr}
        add fp, sp,#4
        sub sp, sp,#68
        str r0, [fp,#-12]
        str r1, [fp,#-16]
        ldr r4, [fp,#-12]
        ldr r0, [r4,#0]
        str r0, [fp,#-8]
        ldr r0, =12
        ldr r1, =0
        muls r3, r0, r1
        str r3, [fp,#-24]
        ldr r0, [fp,#-8]
        ldr r1, [fp,#-24]
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-20]
        ldr r0, [fp,#-20]
        cmp r0,#0
        beq end6
        ldr r0, =.LC11

        bl  printf
        str r0, [fp,#-28]
        ldr r0, =0
        b 
delete_from_end_exit
end6:
        ldr r4, [fp,#-16]
        ldr r0, [r4,#0]
        str r0, [fp,#-8]
        ldr r0, [fp,#-8]
        ldr r1, =0
        adds r3, r0, r1
        str r3, [fp,#-32]
        ldr r0, =12
        ldr r1, =0
        muls r3, r0, r1
        str r3, [fp,#-40]
        ldr r4, [fp,#-32]
        ldr r0, [r4,#0]
        ldr r1, [fp,#-40]
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-36]
        ldr r0, [fp,#-36]
        cmp r0,#0
        beq end7
        ldr r4, [fp,#-16]
        ldr r0, [r4,#0]
        ldr r1, =4
        adds r3, r0, r1
        str r3, [fp,#-44]
        ldr r0, =.LC12
        ldr r4, [fp,#-44]
        ldr r1, [r4,#0]

        bl  printf
        str r0, [fp,#-48]
        ldr r0, [fp,#-8]

        bl  free
        str r0, [fp,#-52]
        ldr r0, =0
        ldr r4, [fp,#-12]
        str r0, [r4,#0]
        ldr r0, =0
        ldr r4, [fp,#-16]
        str r0, [r4,#0]
        ldr r0, =0
        b 
delete_from_end_exit
end7:
        ldr r4, [fp,#-16]
        ldr r0, [r4,#0]
        ldr r1, =4
        adds r3, r0, r1
        str r3, [fp,#-56]
        ldr r0, =.LC13
        ldr r4, [fp,#-56]
        ldr r1, [r4,#0]

        bl  printf
        str r0, [fp,#-60]
        ldr r0, [fp,#-8]
        ldr r1, =0
        adds r3, r0, r1
        str r3, [fp,#-64]
        ldr r4, [fp,#-64]
        ldr r0, [r4,#0]
        ldr r4, [fp,#-16]
        str r0, [r4,#0]
        ldr r4, [fp,#-16]
        ldr r0, [r4,#0]
        ldr r1, =8
        adds r3, r0, r1
        str r3, [fp,#-68]
        ldr r0, =0
        ldr r4, [fp,#-68]
        str r0, [r4,#0]
        ldr r0, [fp,#-8]

        bl  free
        str r0, [fp,#-72]
        ldr r0, =0
        b 
delete_from_end_exit
delete_from_end_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp, 
pc}

.data
.LC14:
        .asciz   "Adding 
to Empty List\n"
.LC15:
        .asciz   "Adding 
to List\n"

.text
.global insert_beginning
insert_beginning:
```

```
        stmfd sp!, {fp,
lr}
        add fp, sp,#4
        sub sp, sp,#68
        str r0, [fp,#-20]
        str r1, [fp,#-24]
        str r2, [fp,#-28]
        ldr r0, =24

        bl  malloc
        str r0, [fp,#-32]
        ldr r0, [fp,#-32]
        str r0, [fp,#-8]
        ldr r0, [fp,#-8]
        ldr r1, =4
        adds r3, r0, r1
        str r3, [fp,#-36]
        ldr r0, [fp,#-20]
        ldr r4, [fp,#-36]
        str r0, [r4,#0]
        ldr r0, =12
        ldr r1, =0
        muls r3, r0, r1
        str r3, [fp,#-44]
        ldr r4, [fp,#-24]
        ldr r0, [r4,#0]
        ldr r1, [fp,#-44]
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-40]
        ldr r0, [fp,#-40]
        cmp r0,#0
        beq else8
        ldr r0, =.LC14

        bl  printf
        str r0, [fp,#-48]
        ldr r0, [fp,#-8]
        ldr r1, =0
        adds r3, r0, r1
        str r3, [fp,#-52]
        ldr r0, =0
        ldr r4, [fp,#-52]
        str r0, [r4,#0]
        ldr r0, [fp,#-8]
        ldr r1, =8
        adds r3, r0, r1
        str r3, [fp,#-56]
        ldr r0, =0
        ldr r4, [fp,#-56]
        str r0, [r4,#0]
        ldr r0, [fp,#-8]
        ldr r4, [fp,#-24]
        str r0, [r4,#0]
        ldr r4, [fp,#-24]
        ldr r0, [r4,#0]
        ldr r4, [fp,#-28]
        str r0, [r4,#0]
        b end8
else8:
        ldr r0, =.LC15

        bl  printf
        str r0, [fp,#-60]
        ldr r0, [fp,#-8]
        str r0, [fp,#-12]
        ldr r0, [fp,#-12]
        ldr r1, =0
        adds r3, r0, r1
        str r3, [fp,#-64]
        ldr r0, =0
        ldr r4, [fp,#-64]
        str r0, [r4,#0]
        ldr r0, [fp,#-12]
        ldr r1, =8
        adds r3, r0, r1
        str r3, [fp,#-68]
        ldr r4, [fp,#-24]
        ldr r0, [r4,#0]
        ldr r4, [fp,#-68]
        str r0, [r4,#0]
        ldr r4, [fp,#-24]
        ldr r0, [r4,#0]
        ldr r1, =0
        adds r3, r0, r1
        str r3, [fp,#-72]
        ldr r0, [fp,#-12]
        ldr r4, [fp,#-72]
        str r0, [r4,#0]
        ldr r0, [fp,#-12]
        ldr r4, [fp,#-24]
        str r0, [r4,#0]
end8:
insert_beginning_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}
```

6.5.7 *TICTACTOE.CPI*

```
int printboard(char board[]){
    printf("|%c|%c|%c|\n", board[0],board[1],board[2]);
    printf("-------\n");
    printf("|%c|%c|%c|\n", board[3],board[4],board[5]);
    printf("-------\n");
    printf("|%c|%c|%c|\n", board[6],board[7],board[8]);
    return 0;
}

int checkrow(char board[], int row){
```

```
    int x1;
    int x2;
    x1 = row + 1;
    x2 = row + 2;
    if (board[row] == board[x1]){
        if (board[x1] == board[x2]){
            if (board[row] != ' '){
                printf("Row win!\n");
                return 1;
            }
        }
    }
    return 0;
}


int checkcol(char board[], int col){
    int x1;
    int x2;
    x1 = col + 3;
    x2 = col + 6;
    if (board[col] == board[x1]){
        if (board[x1] == board[x2]){
            if (board[col] != ' '){
                printf("Column win!\n");
                return 1;
            }
        }
    }
    return 0;
}

int checkboard(char board[]){
    int result;
    int j;
    result = 0;

    for (j = 0; j < 3; j = j + 1){
        result = result + checkrow(board, 3*j) + checkcol(board, j);
    }

    // Check diags
    if (board[0] != ' '){
        if (board[0] == board[4]){
            if (board[4] == board[8]){
                result = 1;
            }
        }
```

```
    }
    if (board[2] != ' '){
        if (board[2] == board[4]){
            if (board[4] == board[6]){
                result = 1;
            }
        }
    }

    return result;
}

char getchar(int p){
    if (p == 1){
        return 'O';
    }
    return 'X';
}


int main()
{
        int player;
        int winner;
        int choice;
    int valid;
    int i;
    int count;
    char board[9];
    char tempc;

    board[0] = ' ';
    board[1] = ' ';
    board[2] = ' ';
    board[3] = ' ';
    board[4] = ' ';
    board[5] = ' ';
    board[6] = ' ';
    board[7] = ' ';
    board[8] = ' ';
    board[9] = ' ';

    printf("Player 1: 'O'\nPlayer 2: 'X'\n\n");
    printf("Valid inputs are 0-9\n\n");

    count = 0;
    winner = 0;
    player = 1;
```

```
    while (winner == 0){
        printboard(board);

        valid = 0;
        while(valid == 0){
            printf("Player %d, enter your move: ", player);
            printf("\n");

            scanf("%d", &choice);

            valid = 1;
            if (choice < 0){ valid = 0; }
            if (choice > 9){ valid = 0; }
            if (valid == 1){
                if (board[choice] != ' '){
                    valid = 0;
                }
            }
        }

        tempc = getchar(player);
        board[choice] = tempc;
        if (checkboard(board) > 0){
            printboard(board);
            printf("Winner is Player %d!\n", player);
            winner = player;
        }

        if (player == 1){
            player = 2;
        } else{
            player = 1;
        }

        count = count + 1;
        if (count >= 9){
            if (winner == 0){
                printf("No one wins!\n");
                winner = -1;
            }
        }
    }
        return 0;
}
```

6.5.8 TICTACTOE.S

```
.data
.LC0:
        .asciz   "Player 1: 'O'\nPlayer 2: 'X'\n\n"
.LC1:
        .asciz   "Valid inputs are 0-9\n\n"
.LC2:
        .asciz   "Player %d, enter your move: "
.LC3:
        .asciz   "\n"
.LC4:
        .asciz   "%d"
.LC5:
        .asciz   "Winner is Player %d!\n"
.LC6:
        .asciz   "No one wins!\n"

.text
.global main
main:
        stmfd sp!, {fp, lr}
        add fp, sp,#4
        sub sp, sp,#320
        ldr r0, =1
        ldr r1, =0
        muls r3, r0, r1
        str r3, [fp,#-48]
        sub r0, fp,#40
        ldr r1, [fp,#-48]
        adds r3, r0, r1
        str r3, [fp,#-60]
        ldrb r0, =32
        ldr r4, [fp,#-60]
        strb r0, [r4,#0]
        ldr r0, =1
        ldr r1, =1
        muls r3, r0, r1
        str r3, [fp,#-64]
        sub r0, fp,#40
        ldr r1, [fp,#-64]
        adds r3, r0, r1
        str r3, [fp,#-76]
        ldrb r0, =32
        ldr r4, [fp,#-76]
        strb r0, [r4,#0]
        ldr r0, =1
        ldr r1, =2
        muls r3, r0, r1
        str r3, [fp,#-80]
        sub r0, fp,#40
        ldr r1, [fp,#-80]
        adds r3, r0, r1
        str r3, [fp,#-92]
        ldrb r0, =32
        ldr r4, [fp,#-92]
        strb r0, [r4,#0]
        ldr r0, =1
        ldr r1, =3
        muls r3, r0, r1
        str r3, [fp,#-96]
        sub r0, fp,#40
        ldr r1, [fp,#-96]
        adds r3, r0, r1
        str r3, [fp,#-108]
        ldrb r0, =32
        ldr r4, [fp,#-108]
        strb r0, [r4,#0]
        ldr r0, =1
        ldr r1, =4
        muls r3, r0, r1
        str r3, [fp,#-112]
        sub r0, fp,#40
        ldr r1, [fp,#-112]
        adds r3, r0, r1
        str r3, [fp,#-124]
        ldrb r0, =32
        ldr r4, [fp,#-124]
        strb r0, [r4,#0]
        ldr r0, =1
        ldr r1, =5
        muls r3, r0, r1
        str r3, [fp,#-128]
        sub r0, fp,#40
        ldr r1, [fp,#-128]
        adds r3, r0, r1
        str r3, [fp,#-140]
        ldrb r0, =32
        ldr r4, [fp,#-140]
        strb r0, [r4,#0]
        ldr r0, =1
        ldr r1, =6
        muls r3, r0, r1
        str r3, [fp,#-144]
        sub r0, fp,#40
        ldr r1, [fp,#-144]
        adds r3, r0, r1
        str r3, [fp,#-156]
        ldrb r0, =32
        ldr r4, [fp,#-156]
        strb r0, [r4,#0]
        ldr r0, =1
        ldr r1, =7
        muls r3, r0, r1
        str r3, [fp,#-160]
        sub r0, fp,#40
        ldr r1, [fp,#-160]
        adds r3, r0, r1
        str r3, [fp,#-172]
        ldrb r0, =32
        ldr r4, [fp,#-172]
        strb r0, [r4,#0]
        ldr r0, =1
        ldr r1, =8
        muls r3, r0, r1
        str r3, [fp,#-176]
        sub r0, fp,#40
        ldr r1, [fp,#-176]
        adds r3, r0, r1
```

```
        str r3, [fp,#-
188]
        ldrb r0, =32
        ldr r4, [fp,#-
188]
        strb r0, [r4,#0]
        ldr r0, =1
        ldr r1, =9
        muls r3, r0, r1
        str r3, [fp,#-
192]
        sub r0, fp,#40
        ldr r1, [fp,#-
192]
        adds r3, r0, r1
        str r3, [fp,#-
204]
        ldrb r0, =32
        ldr r4, [fp,#-
204]
        strb r0, [r4,#0]
        ldr r0, =.LC0

        bl  printf
        str r0, [fp,#-
208]
        ldr r0, =.LC1

        bl  printf
        str r0, [fp,#-
212]
        ldr r0, =0
        str r0, [fp,#-28]
        ldr r0, =0
        str r0, [fp,#-12]
        ldr r0, =1
        str r0, [fp,#-8]
        b loop2_end
loop2_start:
        sub r0, fp,#40

        bl  printboard
        str r0, [fp,#-
216]
        ldr r0, =0
        str r0, [fp,#-20]
        b loop1_end
loop1_start:
        ldr r0, =.LC2
        ldr r1, [fp,#-8]

        bl  printf
        str r0, [fp,#-
220]
        ldr r0, =.LC3

        bl  printf
        str r0, [fp,#-
224]
        ldr r0, =.LC4
        sub r1, fp,#16

        bl  scanf
        str r0, [fp,#-
228]
        ldr r0, =1
        str r0, [fp,#-20]
        ldr r0, [fp,#-16]
        ldr r1, =0
        cmp r0, r1
        movlt r3,#1
        movge r3,#0
        uxtb r3,r3
        str r3, [fp,#-
232]
        ldr r0, [fp,#-
232]

        cmp r0,#0
        beq end1
        ldr r0, =0
        str r0, [fp,#-20]
end1:
        ldr r0, [fp,#-16]
        ldr r1, =9
        cmp r0, r1
        movgt r3,#1
        movle r3,#0
        uxtb r3,r3
        str r3, [fp,#-
236]
        ldr r0, [fp,#-
236]

        cmp r0,#0
        beq end2
        ldr r0, =0
        str r0, [fp,#-20]
end2:
        ldr r0, [fp,#-20]
        ldr r1, =1
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-
240]
        ldr r0, [fp,#-
240]

        cmp r0,#0
        beq end4
        ldr r0, =1
        ldr r1, [fp,#-16]
        muls r3, r0, r1
        str r3, [fp,#-
244]
        sub r0, fp,#40
        ldr r1, [fp,#-
244]

        adds r3, r0, r1
        str r3, [fp,#-
256]
        ldr r4, [fp,#-
256]
        ldrb r0, [r4,#0]
        ldrb r1, =32
        cmp r0, r1
        moveq r3,#0
        movne r3,#1
        uxtb r3,r3
        strb r3, [fp,#-
257]
        ldrb r0, [fp,#-
257]

        cmp r0,#0
        beq end3
        ldr r0, =0
        str r0, [fp,#-20]
end3:
end4:
loop1_end:
        ldr r0, [fp,#-20]
        ldr r1, =0
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
```

```
        uxtb r3,r3
        str r3, [fp,#-
264]
        ldr r0, [fp,#-
264]
        cmp r0,#0
        bne loop1_start
        ldr r0, [fp,#-8]

        bl  getchar
        strb r0, [fp,#-
265]
        ldrb r0, [fp,#-
265]
        strb r0, [fp,#-
41]
        ldr r0, =1
        ldr r1, [fp,#-16]
        muls r3, r0, r1
        str r3, [fp,#-
272]
        sub r0, fp,#40
        ldr r1, [fp,#-
272]
        adds r3, r0, r1
        str r3, [fp,#-
284]
        ldrb r0, [fp,#-
41]
        ldr r4, [fp,#-
284]
        strb r0, [r4,#0]
        sub r0, fp,#40

        bl  checkboard
        str r0, [fp,#-
288]
        ldr r0, [fp,#-
288]
        ldr r1, =0
        cmp r0, r1
        movgt r3,#1
        movle r3,#0
        uxtb r3,r3
        str r3, [fp,#-
292]
        ldr r0, [fp,#-
292]
        cmp r0,#0
        beq end5
        sub r0, fp,#40

        bl  printboard
        str r0, [fp,#-
296]
        ldr r0, =.LC5
        ldr r1, [fp,#-8]

        bl  printf
        str r0, [fp,#-
300]
        ldr r0, [fp,#-8]
        str r0, [fp,#-12]
end5:
        ldr r0, [fp,#-8]
        ldr r1, =1
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-
304]
        ldr r0, [fp,#-
304]
        cmp r0,#0
        beq else6
        ldr r0, =2
        str r0, [fp,#-8]
        b end6
else6:
        ldr r0, =1
        str r0, [fp,#-8]
end6:
        ldr r0, [fp,#-28]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-
308]
        ldr r0, [fp,#-
308]
        str r0, [fp,#-28]
        ldr r0, [fp,#-28]
        ldr r1, =9
        cmp r0, r1
        movge r3,#1
        movlt r3,#0
        uxtb r3,r3
        str r3, [fp,#-
312]
        ldr r0, [fp,#-
312]
        cmp r0,#0
        beq end8
        ldr r0, [fp,#-12]
        ldr r1, =0
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-
316]
        ldr r0, [fp,#-
316]
        cmp r0,#0
        beq end7
        ldr r0, =.LC6

        bl  printf
        str r0, [fp,#-
320]
        ldr r0, =-1
        str r0, [fp,#-12]
end7:
end8:
loop2_end:
        ldr r0, [fp,#-12]
        ldr r1, =0
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-
324]
        ldr r0, [fp,#-
324]
        cmp r0,#0
        bne loop2_start
        ldr r0, =0
        b main_exit
main_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}
```

```
.data

.text
.global getchar
getchar:
        stmfd sp!, {fp,
lr}
        add fp, sp,#4
        sub sp, sp,#8
        str r0, [fp,#-8]
        ldr r0, [fp,#-8]
        ldr r1, =1
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        str r3, [fp,#-12]
        ldr r0, [fp,#-12]
        cmp r0,#0
        beq end9
        ldrb r0, =79
        b getchar_exit
end9:
        ldrb r0, =88
        b getchar_exit
getchar_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}

.data

.text
.global checkboard
checkboard:
        stmfd sp!, {fp,
lr}
        add fp, sp,#4
        sub sp, sp,#144
        str r0, [fp,#-16]
        ldr r0, =0
        str r0, [fp,#-8]
        ldr r0, =0
        str r0, [fp,#-12]
        b loop3_end
loop3_start:
        ldr r0, =3
        ldr r1, [fp,#-12]
        muls r3, r0, r1
        str r3, [fp,#-20]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-20]

        bl  checkrow
        str r0, [fp,#-24]
        ldr r0, [fp,#-8]
        ldr r1, [fp,#-24]
        adds r3, r0, r1
        str r3, [fp,#-28]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-12]

        bl  checkcol
        str r0, [fp,#-32]
        ldr r0, [fp,#-28]
        ldr r1, [fp,#-32]
        adds r3, r0, r1
        str r3, [fp,#-36]
        ldr r0, [fp,#-36]
        str r0, [fp,#-8]
        ldr r0, [fp,#-12]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-40]
        ldr r0, [fp,#-40]
        str r0, [fp,#-12]
loop3_end:
        ldr r0, [fp,#-12]
        ldr r1, =3
        cmp r0, r1
        movlt r3,#1
        movge r3,#0
        uxtb r3,r3
        str r3, [fp,#-44]
        ldr r0, [fp,#-44]
        cmp r0,#0
        bne loop3_start
        ldr r0, =1
        ldr r1, =0
        muls r3, r0, r1
        str r3, [fp,#-48]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-48]
        adds r3, r0, r1
        str r3, [fp,#-52]
        ldr r4, [fp,#-52]
        ldrb r0, [r4,#0]
        ldrb r1, =32
        cmp r0, r1
        moveq r3,#0
        movne r3,#1
        uxtb r3,r3
        strb r3, [fp,#-
53]
        ldrb r0, [fp,#-
53]
        cmp r0,#0
        beq end12
        ldr r0, =1
        ldr r1, =0
        muls r3, r0, r1
        str r3, [fp,#-60]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-60]
        adds r3, r0, r1
        str r3, [fp,#-64]
        ldr r0, =1
        ldr r1, =4
        muls r3, r0, r1
        str r3, [fp,#-68]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-68]
        adds r3, r0, r1
        str r3, [fp,#-72]
        ldr r4, [fp,#-64]
        ldrb r0, [r4,#0]
        ldr r4, [fp,#-72]
        ldrb r1, [r4,#0]
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
73]
        ldrb r0, [fp,#-
73]
        cmp r0,#0
        beq end11
        ldr r0, =1
        ldr r1, =4
        muls r3, r0, r1
        str r3, [fp,#-80]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-80]
```

```
        adds r3, r0, r1
        str r3, [fp,#-84]
        ldr r0, =1
        ldr r1, =8
        muls r3, r0, r1
        str r3, [fp,#-88]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-88]
        adds r3, r0, r1
        str r3, [fp,#-92]
        ldr r4, [fp,#-84]
        ldrb r0, [r4,#0]
        ldr r4, [fp,#-92]
        ldrb r1, [r4,#0]
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
93]
        ldrb r0, [fp,#-
93]
        cmp r0,#0
        beq end10
        ldr r0, =1
        str r0, [fp,#-8]
end10:
end11:
end12:
        ldr r0, =1
        ldr r1, =2
        muls r3, r0, r1
        str r3, [fp,#-
100]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-
100]
        adds r3, r0, r1
        str r3, [fp,#-
104]
        ldr r4, [fp,#-
104]
        ldrb r0, [r4,#0]
        ldrb r1, =32
        cmp r0, r1
        moveq r3,#0
        movne r3,#1
        uxtb r3,r3
        strb r3, [fp,#-
105]
        ldrb r0, [fp,#-
105]
        cmp r0,#0
        beq end15
        ldr r0, =1
        ldr r1, =2
        muls r3, r0, r1
        str r3, [fp,#-
112]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-
112]
        adds r3, r0, r1
        str r3, [fp,#-
116]
        ldr r0, =1
        ldr r1, =4
        muls r3, r0, r1
        str r3, [fp,#-
120]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-
120]
        adds r3, r0, r1
        str r3, [fp,#-
124]
        ldr r4, [fp,#-
116]
        ldrb r0, [r4,#0]
        ldr r4, [fp,#-
124]
        ldrb r1, [r4,#0]
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
125]
        ldrb r0, [fp,#-
125]
        cmp r0,#0
        beq end14
        ldr r0, =1
        ldr r1, =4
        muls r3, r0, r1
        str r3, [fp,#-
132]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-
132]
        adds r3, r0, r1
        str r3, [fp,#-
136]
        ldr r0, =1
        ldr r1, =6
        muls r3, r0, r1
        str r3, [fp,#-
140]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-
140]
        adds r3, r0, r1
        str r3, [fp,#-
144]
        ldr r4, [fp,#-
136]
        ldrb r0, [r4,#0]
        ldr r4, [fp,#-
144]
        ldrb r1, [r4,#0]
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-
145]
        ldrb r0, [fp,#-
145]
        cmp r0,#0
        beq end13
        ldr r0, =1
        str r0, [fp,#-8]
end13:
end14:
end15:
        ldr r0, [fp,#-8]
        b checkboard_exit
checkboard_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}

.data
```

```
.LC7:
        .asciz  "Column win!\n"

.text
.global checkcol
checkcol:
        stmfd sp!, {fp, lr}
        add fp, sp,#4
        sub sp, sp,#80
        str r0, [fp,#-16]
        str r1, [fp,#-20]
        ldr r0, [fp,#-20]
        ldr r1, =3
        adds r3, r0, r1
        str r3, [fp,#-24]
        ldr r0, [fp,#-24]
        str r0, [fp,#-8]
        ldr r0, [fp,#-20]
        ldr r1, =6
        adds r3, r0, r1
        str r3, [fp,#-28]
        ldr r0, [fp,#-28]
        str r0, [fp,#-12]
        ldr r0, =1
        ldr r1, [fp,#-20]
        muls r3, r0, r1
        str r3, [fp,#-32]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-32]
        adds r3, r0, r1
        str r3, [fp,#-36]
        ldr r0, =1
        ldr r1, [fp,#-8]
        muls r3, r0, r1
        str r3, [fp,#-40]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-40]
        adds r3, r0, r1
        str r3, [fp,#-44]
        ldr r4, [fp,#-36]
        ldrb r0, [r4,#0]
        ldr r4, [fp,#-44]
        ldrb r1, [r4,#0]
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-45]
        ldrb r0, [fp,#-45]
        cmp r0,#0
        beq end18
        ldr r0, =1
        ldr r1, [fp,#-8]
        muls r3, r0, r1
        str r3, [fp,#-52]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-52]
        adds r3, r0, r1
        str r3, [fp,#-56]
        ldr r0, =1
        ldr r1, [fp,#-12]
        muls r3, r0, r1
        str r3, [fp,#-60]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-60]
        adds r3, r0, r1
        str r3, [fp,#-64]
        ldr r4, [fp,#-56]
        ldrb r0, [r4,#0]
        ldr r4, [fp,#-64]
        ldrb r1, [r4,#0]
        cmp r0, r1
        moveq r3,#1
        movne r3,#0
        uxtb r3,r3
        strb r3, [fp,#-65]
        ldrb r0, [fp,#-65]
        cmp r0,#0
        beq end17
        ldr r0, =1
        ldr r1, [fp,#-20]
        muls r3, r0, r1
        str r3, [fp,#-72]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-72]
        adds r3, r0, r1
        str r3, [fp,#-76]
        ldr r4, [fp,#-76]
        ldrb r0, [r4,#0]
        ldrb r1, =32
        cmp r0, r1
        moveq r3,#0
        movne r3,#1
        uxtb r3,r3
        strb r3, [fp,#-77]
        ldrb r0, [fp,#-77]
        cmp r0,#0
        beq end16
        ldr r0, =.LC7

        bl  printf
        str r0, [fp,#-84]
        ldr r0, =1
        b checkcol_exit
end16:
end17:
end18:
        ldr r0, =0
        b checkcol_exit
checkcol_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp, pc}

.data
.LC8:
        .asciz  "Row win!\n"

.text
.global checkrow
checkrow:
        stmfd sp!, {fp, lr}
        add fp, sp,#4
        sub sp, sp,#80
        str r0, [fp,#-16]
        str r1, [fp,#-20]
        ldr r0, [fp,#-20]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-24]
        ldr r0, [fp,#-24]
        str r0, [fp,#-8]
        ldr r0, [fp,#-20]
        ldr r1, =2
```

```
adds r3, r0, r1
str r3, [fp,#-28]
ldr r0, [fp,#-28]
str r0, [fp,#-12]
ldr r0, =1
ldr r1, [fp,#-20]
muls r3, r0, r1
str r3, [fp,#-32]
ldr r0, [fp,#-16]
ldr r1, [fp,#-32]
adds r3, r0, r1
str r3, [fp,#-36]
ldr r0, =1
ldr r1, [fp,#-8]
muls r3, r0, r1
str r3, [fp,#-40]
ldr r0, [fp,#-16]
ldr r1, [fp,#-40]
adds r3, r0, r1
str r3, [fp,#-44]
ldr r4, [fp,#-36]
ldrb r0, [r4,#0]
ldr r4, [fp,#-44]
ldrb r1, [r4,#0]
cmp r0, r1
moveq r3,#1
movne r3,#0
uxtb r3,r3
strb r3, [fp,#-
45]
ldrb r0, [fp,#-
45]
cmp r0,#0
beq end21
ldr r0, =1
ldr r1, [fp,#-8]
muls r3, r0, r1
str r3, [fp,#-52]
ldr r0, [fp,#-16]
ldr r1, [fp,#-52]
adds r3, r0, r1
str r3, [fp,#-56]
ldr r0, =1
ldr r1, [fp,#-12]
muls r3, r0, r1
str r3, [fp,#-60]
ldr r0, [fp,#-16]
ldr r1, [fp,#-60]
adds r3, r0, r1
str r3, [fp,#-64]
ldr r4, [fp,#-56]
ldrb r0, [r4,#0]
ldr r4, [fp,#-64]
ldrb r1, [r4,#0]
cmp r0, r1
moveq r3,#1
movne r3,#0
uxtb r3,r3
strb r3, [fp,#-
65]
ldrb r0, [fp,#-
65]
cmp r0,#0
beq end20
ldr r0, =1
ldr r1, [fp,#-20]
muls r3, r0, r1
str r3, [fp,#-72]
ldr r0, [fp,#-16]
ldr r1, [fp,#-72]
adds r3, r0, r1
str r3, [fp,#-76]
ldr r4, [fp,#-76]
ldrb r0, [r4,#0]
ldrb r1, =32
cmp r0, r1
moveq r3,#0
movne r3,#1
uxtb r3,r3
strb r3, [fp,#-
77]
ldrb r0, [fp,#-
77]
cmp r0,#0
beq end19
ldr r0, =.LC8

bl  printf
str r0, [fp,#-84]
ldr r0, =1
b checkrow_exit
end19:
end20:
end21:
ldr r0, =0
b checkrow_exit
checkrow_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}

.data
.LC9:
        .asciz
"|%c|%c|%c|\n"
.LC10:
        .asciz   "-------
\n"
.LC11:
        .asciz
"|%c|%c|%c|\n"
.LC12:
        .asciz   "-------
\n"
.LC13:
        .asciz
"|%c|%c|%c|\n"

.text
.global printboard
printboard:
        stmfd sp!, {fp,
lr}
        add fp, sp,#4
        sub sp, sp,#96
        str r0, [fp,#-8]
        ldr r0, =1
        ldr r1, =2
        muls r3, r0, r1
        str r3, [fp,#-12]
        ldr r0, [fp,#-8]
        ldr r1, [fp,#-12]
        adds r3, r0, r1
        str r3, [fp,#-16]
        ldr r0, =1
        ldr r1, =1
        muls r3, r0, r1
        str r3, [fp,#-20]
        ldr r0, [fp,#-8]
        ldr r1, [fp,#-20]
        adds r3, r0, r1
        str r3, [fp,#-24]
        ldr r0, =1
        ldr r1, =0
```

```
muls r3, r0, r1
str r3, [fp,#-28]
ldr r0, [fp,#-8]
ldr r1, [fp,#-28]
adds r3, r0, r1
str r3, [fp,#-32]
ldr r0, =.LC9
ldr r4, [fp,#-32]
ldrb r1, [r4,#0]
ldr r4, [fp,#-24]
ldrb r2, [r4,#0]
ldr r4, [fp,#-16]
ldrb r3, [r4,#0]

bl  printf
str r0, [fp,#-36]
ldr r0, =.LC10

bl  printf
str r0, [fp,#-40]
ldr r0, =1
ldr r1, =5
muls r3, r0, r1
str r3, [fp,#-44]
ldr r0, [fp,#-8]
ldr r1, [fp,#-44]
adds r3, r0, r1
str r3, [fp,#-48]
ldr r0, =1
ldr r1, =4
muls r3, r0, r1
str r3, [fp,#-52]
ldr r0, [fp,#-8]
ldr r1, [fp,#-52]
adds r3, r0, r1
str r3, [fp,#-56]
ldr r0, =1
ldr r1, =3
muls r3, r0, r1
str r3, [fp,#-60]
ldr r0, [fp,#-8]
ldr r1, [fp,#-60]
adds r3, r0, r1
str r3, [fp,#-64]
ldr r0, =.LC11
ldr r4, [fp,#-64]
ldrb r1, [r4,#0]
ldr r4, [fp,#-56]
ldrb r2, [r4,#0]
ldr r4, [fp,#-48]
ldrb r3, [r4,#0]

bl  printf
str r0, [fp,#-68]
ldr r0, =.LC12

bl  printf
str r0, [fp,#-72]
ldr r0, =1
ldr r1, =8
muls r3, r0, r1
str r3, [fp,#-76]
ldr r0, [fp,#-8]
ldr r1, [fp,#-76]
adds r3, r0, r1
str r3, [fp,#-80]
ldr r0, =1
ldr r1, =7
muls r3, r0, r1
str r3, [fp,#-84]
ldr r0, [fp,#-8]
ldr r1, [fp,#-84]
adds r3, r0, r1
str r3, [fp,#-88]
ldr r0, =1
ldr r1, =6
muls r3, r0, r1
str r3, [fp,#-92]
ldr r0, [fp,#-8]
ldr r1, [fp,#-92]
adds r3, r0, r1
str r3, [fp,#-96]
ldr r0, =.LC13
ldr r4, [fp,#-96]
ldrb r1, [r4,#0]
ldr r4, [fp,#-88]
ldrb r2, [r4,#0]
ldr r4, [fp,#-80]
ldrb r3, [r4,#0]

bl  printf
str r0, [fp,#-
100]
ldr r0, =0
b printboard_exit
printboard_exit:
sub sp, fp, #4
ldmfd sp!, {fp,
pc}
```

6.5.9 VARIABLEARRAY.CPI

```
#include <stdio.h>

int fun(int n) {
  int c;
  int a[n];
  char b[n];
  int sum;
  c = 0;


  while(c < n) {
```

```
    a[c] = c;
    b[c] = 'a' + c;
    c = c+ 1;
  }

  c = 0;
  sum =0;
  while(c < n) {
    printf("a[%d] = %d\t",c,a[c]);
    printf("b[%d] = %c\n",c,b[c]);
    sum = sum + a[c];
    c = c+ 1;
  }
  return sum;
}


int main() {
  return fun(10);
}
```

6.5.10 VARIABLEARRAY.S

```
.data

.text
.global main
main:
        stmfd sp!, {fp,
lr}
        add fp, sp,#4
        sub sp, sp,#4
        ldr r0, =10

        bl  fun
        str r0, [fp,#-8]
        ldr r0, [fp,#-8]
        b main_exit
main_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}

.data
.LC0:
        .asciz   "a[%d] =
%d\t"
.LC1:
        .asciz   "b[%d] =
%c\n"

.text
.global fun
fun:
        stmfd sp!, {fp,
lr}
        add fp, sp,#4
        sub sp, sp,#100
        str r0, [fp,#-24]
        ldr r0, =4
        ldr r1, [fp,#-24]
        muls r3, r0, r1
        str r3, [fp,#-
104]
        ldr r0, [fp,#-
104]
        lsr r1,r0,#2
        lsl r1,r1,#2
        cmp r1,r0
        movne r0,#4
        moveq r0,#0
        add r3,r0,r1
        sub sp, sp,r3
        mov r0,sp
        str r0, [fp,#-12]
        ldr r0, =1
        ldr r1, [fp,#-24]
        muls r3, r0, r1
        strb r3, [fp,#-
97]
        ldrb r0, [fp,#-
97]
        lsr r1,r0,#2
        lsl r1,r1,#2
        cmp r1,r0
        movne r0,#4
        moveq r0,#0
        add r3,r0,r1
        sub sp, sp,r3
        mov r0,sp
        str r0, [fp,#-16]
        ldr r0, =0
        str r0, [fp,#-8]
        b loop1_end
loop1_start:
```

```
        ldr r0, =4
        ldr r1, [fp,#-8]
        muls r3, r0, r1
        str r3, [fp,#-28]
        ldr r0, [fp,#-12]
        ldr r1, [fp,#-28]
        adds r3, r0, r1
        str r3, [fp,#-32]
        ldr r0, [fp,#-8]
        ldr r4, [fp,#-32]
        str r0, [r4,#0]
        ldrb r0, =97
        ldr r1, [fp,#-8]
        adds r3, r0, r1
        str r3, [fp,#-36]
        ldr r0, =1
        ldr r1, [fp,#-8]
        muls r3, r0, r1
        str r3, [fp,#-40]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-40]
        adds r3, r0, r1
        str r3, [fp,#-44]
        ldr r0, [fp,#-36]
        ldr r4, [fp,#-44]
        strb r0, [r4,#0]
        ldr r0, [fp,#-8]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-48]
        ldr r0, [fp,#-48]
        str r0, [fp,#-8]
loop1_end:
        ldr r0, [fp,#-8]
        ldr r1, [fp,#-24]
        cmp r0, r1
        movlt r3,#1
        movge r3,#0
        uxtb r3,r3
        str r3, [fp,#-52]
        ldr r0, [fp,#-52]
        cmp r0,#0
        bne loop1_start
        ldr r0, =0
        str r0, [fp,#-8]
        ldr r0, =0
        str r0, [fp,#-20]
        b loop2_end
loop2_start:
        ldr r0, =4
        ldr r1, [fp,#-8]
        muls r3, r0, r1
        str r3, [fp,#-56]
        ldr r0, [fp,#-12]
        ldr r1, [fp,#-56]
        adds r3, r0, r1
        str r3, [fp,#-60]
        ldr r0, =.LC0
        ldr r1, [fp,#-8]
        ldr r4, [fp,#-60]
        ldr r2, [r4,#0]

        bl  printf
        str r0, [fp,#-64]
        ldr r0, =1
        ldr r1, [fp,#-8]
        muls r3, r0, r1
        str r3, [fp,#-68]
        ldr r0, [fp,#-16]
        ldr r1, [fp,#-68]
        adds r3, r0, r1
        str r3, [fp,#-72]
        ldr r0, =.LC1
        ldr r1, [fp,#-8]
        ldr r4, [fp,#-72]
        ldrb r2, [r4,#0]

        bl  printf
        str r0, [fp,#-76]
        ldr r0, =4
        ldr r1, [fp,#-8]
        muls r3, r0, r1
        str r3, [fp,#-80]
        ldr r0, [fp,#-12]
        ldr r1, [fp,#-80]
        adds r3, r0, r1
        str r3, [fp,#-84]
        ldr r0, [fp,#-20]
        ldr r4, [fp,#-84]
        ldr r1, [r4,#0]
        adds r3, r0, r1
        str r3, [fp,#-88]
        ldr r0, [fp,#-88]
        str r0, [fp,#-20]
        ldr r0, [fp,#-8]
        ldr r1, =1
        adds r3, r0, r1
        str r3, [fp,#-92]
        ldr r0, [fp,#-92]
        str r0, [fp,#-8]
loop2_end:
        ldr r0, [fp,#-8]
        ldr r1, [fp,#-24]
        cmp r0, r1
        movlt r3,#1
        movge r3,#0
        uxtb r3,r3
        str r3, [fp,#-96]
        ldr r0, [fp,#-96]
        cmp r0,#0
        bne loop2_start
        ldr r0, [fp,#-20]
        b fun_exit
fun_exit:
        sub sp, fp, #4
        ldmfd sp!, {fp,
pc}
```

7 Lessons Learned

7.1 Edward Garcia

7.1.1 Lessons Learned

Pattern matching should be a feature available in all languages. Strongly typed, functional constructs can make code a lot cleaner to read/interpret and reduce the probability of introducing bugs. Also the time spent upfront to make regression tests is worth the effort. Early on, it is fairly easy to introduce new features and not worry about breaking anything. However, near the end of the term when we implementing our last features, we would often find unintended consequences of changing code in other features we had implemented. Regression tests and Git were the key to finding the source of the problem.

7.1.2 ADVICE FOR FUTURE STUDENTS

While reviewing Ocaml in class and doing homework assignments was beneficial, it took me a while to get a good grip of the language. The greatest thing that helped was applying Ocaml to solve problems and viewing example source code. The website http://ocaml.org/learn/tutorials/99problems.html has 99 problems that you can attempt and provides solutions to compare against. Also during the semester, I developed an interest in the LLVM open source project (http://llvm.org/). There a variety of front ends and back ends that are being developed for the project and if I had to do it all over, I would target the LLVM intermediate representation.

7.2 Sean Yeh

7.2.1 LESSONS LEARNED

Next time I will not write test suite script in BASH. Nevertheless, the testing framework turned out pretty well. I also learned that spending some time on the design will save much time and frustration later.

7.2.2 ADVICE FOR FUTURE STUDENTS

Try to set up an automatic testing framework as early as you can; a good testing framework will save a lot of your time later on in the project. Also, make sure you familiarize yourself with a version control system (such as git). Otherwise, just have fun!

7.3 NAVEEN REVANNA

7.3.1 LESSONS LEARNED

Spend sufficient time in deciding a scalable architecture at early stages. This can save a lot of time when more and more features gets added.

Don't trust your developer self. A single line can indeed break the whole system. Don't be sure until you test it.

Document code sufficiently. A week later ocaml code becomes cryptic to oneself.

A good test infrastructure can save you loads of time.

ADVICE FOR FUTURE STUDENTS Even though rework on architecture is inevitable as more and more features are added, good time spent on a scalable architecture will save significant time as new things get added. So start early. Don't downplay the importance of a good version control system and bug tracking system. Github should be a good candidate. Ocaml is like a wild horse, you can have a good ride once you tame it.

7.4 NIKET KANDYA

7.4.1 LESSONS LEARNED

Time spent on good design is time saved. Functional Programming is cool and is a neat idea. Compilers are fun and not magic after all :).

ADVICE FOR FUTURE STUDENTS: Start early and try to think about the all the infrastructure you might need right from the start; but importantly, without getting lost in details and focus. Meeting an advisor is definitely helpful to keep you on the right track. Keep working regularly on the project as things can change very dramatically in a short span, especially with a language like OCaml. Take a project which you are passionate about.

8 APPENDIX

8.1 AST.ML

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq |
Lor | Land
type resolve = Dot | Ind

type expr =
    Literal of int
  | String of string
  | Addrof of expr
  | Negof of expr
  | ConstCh of string
  | Id of string
  | MultiId of expr * resolve * expr
  | Pointer of expr
  | Array of expr * expr
  | Binop of expr * op * expr
  | Assign of expr * expr
  | Call of string * expr list
  | Null
```

```
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

type cpitypes = Void | Int | Char | Ptr | Arr of expr | Struct of string | Err

type var_decl = {
  vname: string;
  vtype: cpitypes list;
}

type struct_decl = {
  sname: string;
  smembers: var_decl list
}

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

type program = {
  sdecls : struct_decl list;
  gdecls : var_decl list;
  fdecls : func_decl list
}
```

8.2 BYTECODE.ML

```
open Ast

type atom =
    Lit of int    (*  literal *)
  | Cchar of char
  | Sstr of string * string (* Sstr(name, label) *)
  | Lvar of int * int(* Lvar(offset,size) *)
  | Gvar of string * int (* Global var (name,size) *)
  | Pntr of atom * int (* Pntr(addr,size) *)
```

```
  | Addr of atom
  | Neg  of atom
  | Debug of string

type bstmt =
    Atom of atom
  | VarArr of atom * atom
  | Rval of atom
  | BinEval of atom * atom * Ast.op * atom (*Binary evaluation *)
  | BinRes of cpitypes list
  | Assgmt of atom * atom
  | Fcall of string * atom list * atom
  | Branch of string
  | Predicate of atom * bool * string (* (var_to_check, jump_on_what? , label)*)
  | Label of string

type prog =
  Fstart of string * atom list * bstmt list * int (*start of a function*)
  | Global of atom list
```

8.3 COMPILE.ML

```
open Sast
open Ast
open Bytecode
open Debug
open Printexc

module StringMap = Map.Make(String)

let err str = raise(Failure("Compile: "^ str));;

let rec get_size_type sindex = function
|[] ->   raise Exit
| hd::tl ->
  (match hd with
    Void -> 0
  | Char -> 1
  | Int
  | Ptr -> 4
  | Arr(sz) -> (match sz with
        Literal(i) -> i * (get_size_type sindex tl)
        | Id(id) -> get_size_type sindex [Ptr]
        | _ -> err "lit_to_num: unexpected")
  | Struct(sname) -> (StringMap.find sname sindex).size
  | _ -> err "Requesting size of wrong type");;

let get_atom = function
```

```
      Atom (atm) -> atm
    | BinEval  (dst, var1, op, var2) -> dst
    | Fcall (fname, args,ret ) -> ret
    | Assgmt (dst, src) -> dst
    | Label (a)-> err ("Unexpected: Label-> " ^ a)
    | Predicate (_, _, _)-> err "Unexpected: Predicate"
    | Branch _-> err "Unexpected: Branch"
    | BinRes(ty) -> err ("Unexpected: BinRes " ^
      dbg_str_of_typs (List.hd ty))
    |Rval _ -> err "Unexpected: Rval"
    | VarArr(_,_) -> err "Unexpected: VarArr";;

let build_global_idx map = StringMap.empty;;
let gl_atm a = get_atom(List.hd ( List.rev a));;

(* Calucates the offset for a variable type based on alignment rules
 * i.e char does not require any alignment. All other current datatypes require
 * alignment *)
let calc_offset sidx offset typlst =
  let align_size = 4 in
  let offset = offset + get_size_type sidx typlst in
    match (List.hd typlst) with
      Char -> offset
    | _ ->  align_size * int_of_float(ceil ((float_of_int offset ) /.(float_of_int
align_size)));;


(* This is to change the type of a input array to Ptr type if its in the formal
 * list. i.e for void foo(int a[]), a will be considered as a Pointer which will
 * point to the array in the caller function *)
let rec modify_formal_lst = function
    [] -> []
    | hd :: tl -> ( (match List.hd (hd.vtype) with
        Arr(_)-> { hd with vtype = Ptr :: List.tl hd.vtype }
        |  _ -> hd ) :: (modify_formal_lst tl));;


(* If its a local variable sized array declaration, then it should
 * considered as a Pointer type and memory allocated accordingly.*)
let rec modify_local_lst  = function
    [] -> []
    | hd :: tl -> ( (match List.hd (hd.vtype) with
        Arr(s)-> (match s with
              Id(id) -> { hd with vtype = Ptr :: List.tl hd.vtype }
              | _ -> hd)
        |  _ -> hd ) :: (modify_local_lst tl));;

(* The optional parameter rev is to signify if the index should be build top
```

```
 * down or bottom up based on if it is a struct index or local index.
 * For struct_index, rev=1
 *)
let rec build_local_idx map sidx offset ?(rev =0) = (function
    [] -> map
  | hd:: tl ->
    offset := (calc_offset sidx !offset hd.vtype);
    build_local_idx ~rev:rev
    ( StringMap.add hd.vname
      {
        offset = !offset - (if rev =0 then rev else (get_size_type sidx hd.vtype));
        typ = hd.vtype
      } map
    )
    sidx offset tl);;


(* Translate a program in AST form into a bytecode program.  Throw an
 *   exception if something is wrong, e.g., a reference to an unknown
 *   variable or function *)
let translate prog =
let structs = prog.sdecls
  and globals = prog.gdecls
  and functions = prog.fdecls in
  let count_loop = ref 0
  and count_mem = ref (-1)
  and count_ifelse = ref 0
  and count_label = ref 0 in

(* Allocate "addresses" for each global variable *)
(* TODO Code generation for globals *)
let global_indexes = build_global_idx globals in
(* Build structure specific symbol table*)
let struct_indexes = List.fold_left
  (fun map stct ->
    let soffset = ref 0 in
      let index = build_local_idx ~rev:1
        StringMap.empty map soffset (List.rev stct.smembers) in
        (
          StringMap.add stct.sname
            {
              size = !soffset;
              memb_index = index
            } map
        )
  )
  StringMap.empty structs
in
```

```
let f_index = List.fold_left
  (fun map fdecl ->
    let rec var_to_lst ind = function
      [] -> []
    (*TODO Check correct values*)
    | hd :: tl -> ( {offset =0; typ = hd.vtype} :: (var_to_lst (ind+1) tl)) in
      StringMap.add fdecl.fname
      {
        param = (var_to_lst 0 fdecl.formals);
        ret_ty = fdecl.ret
      }
      map
  )
  StringMap.empty functions
in

(* Add the built-in-function printf, scanf to the function indexes *)
let f2_index =
  StringMap.add "printf"
  {
    param = [];
    ret_ty = [Int]
  }
  f_index
in

let f3_index =
  StringMap.add "scanf"
  {
    param = [];
    ret_ty = [Int]
  }
  f2_index
in

let f4_index =
  StringMap.add "malloc"
  {
    param = [];
    ret_ty = [Int]
  }
  f3_index
in

let function_indexes =
  StringMap.add "free"
  {
```

```
    param = [];
    ret_ty = [Int]
  }
  f4_index
in
(* Translate a function in AST form into a list of bytecode statements *)
let translate env fdecl=
  let curr_offset = ref 0 in

  let env =
    {
      env with local_index =
        (build_local_idx StringMap.empty env.struct_index curr_offset
        ( (modify_local_lst fdecl.locals)
        @ (modify_formal_lst fdecl.formals)))
    }
    in
  let add_temp typlst =
    curr_offset := (calc_offset env.struct_index !curr_offset typlst);
    Lvar(!curr_offset,(get_size_type env.struct_index typlst))
    in
  let get_func_entry name =
    try StringMap.find name env.function_index
    with Not_found -> err ("Function not found : " ^ name)
    in
  let get_type_varname table varname =
    try (StringMap.find varname table).typ
    with Not_found -> err ("Varname not found: "^varname^(string_of_int
    (StringMap.cardinal table)))
    in
  let get_size_varname table varname =
    get_size_type env.struct_index (get_type_varname table varname)
    in
  let get_lvar_varname table strict var =
    try Lvar((StringMap.find var table).offset, (get_size_varname table var))
    with Not_found ->
      try
        if strict = 0 then
          Gvar(var,(get_size_varname table var))
        else raise Not_found
      with Not_found -> err (var ^": Not found")
    in
  let get_ptrsize_type typlst =
    get_size_type env.struct_index (List.tl typlst)
    in
  let get_ptrsize_varname table varname =
    get_size_type env.struct_index (List.tl (get_type_varname table varname))
    in
```

```
let get_binres_type e =
  match List.hd e with
    BinRes(typ) -> typ
  | _ -> err "Unexpted type: Expected BinRes"
  in
let gen_binres_type typ =
  [BinRes(typ)]
  in
let get_dom_type typ1 typ2 =
  ( match List.hd typ1 with
    Ptr
  | Arr(_) -> typ1
  | _ -> (match List.hd typ2 with
          Ptr | Arr(_) -> typ2
          | _ -> (if (get_size_type env.struct_index typ1) <=
                    (get_size_type env.struct_index typ2)
                     then typ2 else typ1)
         )
  )
  in
let raise_error_atom a =
  match a with
    Lit (i) -> err ("Literal " ^ string_of_int i)
  | Cchar(ch) -> err "Const Char"
  | Sstr (s, l) -> err ("StringConst "^s)
  | Lvar (o,s) -> err " Lvar"
  | Gvar (_,_) -> err "Gvar"
  | Pntr (_,_) -> err "Pntr"
  | Addr (_) -> err "Addr"
  | Debug (_)  -> err "Debug"
  | Neg (_) -> err "Negative"
  in
let rec conv2_byt_lvar = function
    [] -> []
  | hd::tl -> let entry = StringMap.find hd.vname env.local_index in
    Lvar(entry.offset, (get_size_type env.struct_index entry.typ))
              :: (conv2_byt_lvar tl)
  in
let get_loop_label num = "loop" ^ match num with
    0 -> string_of_int (count_loop := !count_loop + 1; !count_loop) ^ "_start"
  | 1 -> string_of_int !count_loop ^ "_end"
  | _ -> ""
  in
let get_ifelse_label num =
  match num with
    0 -> "else" ^ string_of_int (count_ifelse := !count_ifelse + 1; !count_ifelse)
  | 1 -> "end" ^ string_of_int !count_ifelse
  | _ -> ""
```

```
  in
let gen_atom atm =
  [Atom (atm)]
  in
let rec get_off_lvar lvar =
  match lvar with
    Lvar(o,s) -> Lit o
  | Addr(l) -> get_off_lvar l
  | _ as a -> raise_error_atom a
  in
let incr_by_ptrsz exp incrsz tmp =
   [BinEval (tmp, (Lit incrsz), Mult, (gl_atm exp))]
  in
let get_struct_table stct =
  (try (StringMap.find stct env.struct_index).memb_index
   with Not_found -> err (" struct " ^ stct ^ " is not a type"))
  in
let gen_addr_lst v1 = v1 @
  gen_atom (Addr(gl_atm v1))
  in
let add_base_offset btyp baddr off =
  let v3 = add_temp btyp in
  let v4 = get_ptrsize_type btyp in
  [BinEval (v3,baddr,Add,off)] @ (gen_atom (Pntr(v3,v4)))
  in
let rec gen_vararr = function
  [] -> []
  | hd :: tl -> (match List.hd (hd.vtype) with
      Arr(s)-> (match s with
            Id(id) -> let tmp =
              add_temp (List.tl hd.vtype)
              in
             (incr_by_ptrsz
            (gen_atom (get_lvar_varname env.local_index 0 id))
            (get_ptrsize_type hd.vtype) tmp) @
              [VarArr((get_lvar_varname env.local_index 0 hd.vname),
                   tmp)]
            | _ -> [])
      |  _ -> []) @ (gen_vararr tl)
  in

  let binop_rest v1 v2 v1binres v2binres binres v3 op=
    (gen_binres_type binres) @ (gen_atom v3) @ (List.tl v1) @
              (List.tl v2) @
              (match List.hd binres with
              Ptr | Arr(_) ->
                  (match List.hd v1binres with
                    Ptr | Arr(_) -> (let tmp = (add_temp v2binres) in
```

```
                        (incr_by_ptrsz v2 (get_size_type env.struct_index
                        (List.tl v1binres)) tmp) @
                        [BinEval (v3 ,(gl_atm v1), op, tmp)])
                       | _ -> (match List.hd v2binres with
                          Ptr | Arr(_) ->
                           let tmp = ((add_temp v1binres)) in
                           (incr_by_ptrsz v1 (get_size_type env.struct_index
                           (List.tl v2binres)) tmp) @
                           [BinEval (v3 ,tmp, op,(gl_atm v2))]
                          | _ -> err "Cannot reach here" )
                       )
                 | _ -> [BinEval (v3 ,(gl_atm v1), op,
                 (gl_atm v2))])
    in
    (* Advantage of using bytecode: While implementing && and ||
     * It was easier to define the login in a slightly higher level
     * language than assembly *)
    let binop_logical  v1 v2 res op = let opvalue = (match op with
        Lor -> true
        | Land -> false
        | _ -> err "Logical only")in
        let endlbl =
      "lend" ^ string_of_int (
        count_label := !count_label + 1;
       !count_label) in
        (gen_binres_type [Int]) @
        [Assgmt (res,Lit(if opvalue then 1 else 0))] @ v1 @
        [Predicate ((gl_atm v1), opvalue, endlbl)] @ v2 @
        [Predicate ((gl_atm v2), false, endlbl)] @
        [Assgmt (res,Lit(if opvalue then 0 else 1))] @ [Label endlbl] @
        gen_atom(res)
    in
let rec expr ?(table = env.local_index) ?(strict=0) = function
        Literal i -> (gen_binres_type [Int]) @ gen_atom (Lit i)
      | String s ->
                let lbl = incr count_mem; ".LC" ^
                (string_of_int !count_mem) in
                (gen_binres_type [Ptr;Char]) @ gen_atom(Sstr(s, lbl))
      | ConstCh(ch) -> (gen_binres_type [Char]) @ gen_atom(Cchar(ch.[1]))
      | Id s ->
                let retyp = get_type_varname table s in
                let v1 = (gen_binres_type(retyp)) @
                         gen_atom(get_lvar_varname table strict s) in
                (match List.hd retyp with
                        Arr(_) -> gen_addr_lst v1
                        | _ -> v1)
      | MultiId(fexpr,Ind, e) -> expr (MultiId(Pointer(fexpr), Dot, e))
      | MultiId(fexpr,Dot,e) ->
```

```
          let v1 = expr fexpr in
          let tab = (match List.hd (get_binres_type v1) with
            Struct(s) -> get_struct_table s
            | _ -> err "Must be a struct") in
          let v2 = expr ~table:tab ~strict:1 e in
          let offset = (match gl_atm v2 with
             Lvar(o,s) -> List.rev(List.tl(List.rev v2)) @
             gen_atom (Lit o)
             | Pntr(b,s) -> (*This will an array *)
                (match (List.nth (List.rev v2) 1) with
                  BinEval(dst,op1,op,op2) ->
                    (List.rev(List.tl(List.tl(List.rev v2)))) @
                    [BinEval(dst,(get_off_lvar op1),Add,op2)]
                    @ gen_atom dst
                  | _ -> err "Array was expected: MultiId")

             | _ -> err "Unexpected type in MultiId") in
          let baddr = (match gl_atm v1 with
                  Lvar(o,s) as l -> Addr(l)
                  | Pntr(b,s) -> b
                  | _ -> err "Unexpected type in MultiId") in
                  List.rev(List.tl(List.rev offset))
                  @ (add_base_offset ( List.hd (get_binres_type offset)
                  ::(get_binres_type offset))
                  baddr (gl_atm offset))
| Binop (e1, op, e2) -> let v1 = expr e1
                          and v2 = expr e2 in
          let v1binres = get_binres_type v1
          and v2binres = get_binres_type v2 in
          let binres = get_dom_type v1binres v2binres in
          let res = (add_temp binres) in
          (match op with
            Lor |Land -> binop_logical v1 v2 res op
            | _ ->  binop_rest v1 v2 v1binres v2binres binres res op
               )
| Assign (s, e) ->
               let v1 = (expr e) and v2 = (expr s)
               in (gen_binres_type (get_binres_type v2))
               @ v1 @ v2 @
          [Assgmt ((gl_atm v2),gl_atm v1)]
| Call (fname, actuals) ->
          let param = List.map expr (List.rev actuals)
          and rettyp = (get_func_entry fname).ret_ty in
          let ret = (add_temp rettyp ) in
          (gen_binres_type rettyp)@
          (gen_atom ret) @ List.concat param @
          [Fcall (fname,List.rev
          (List.map (fun par -> gl_atm par) param)
```

```
              ,ret)]
        | Pointer(e) -> let v1 = expr e in
                         let binresv1 = (get_binres_type v1) in
                         (gen_binres_type (List.tl binresv1)) @
                        v1 @ gen_atom (Pntr( (gl_atm v1),
                        (get_ptrsize_type binresv1)))
        | Array(base,e) -> let v1 = expr e in
                                 let v2 = expr base in
                                 let off = add_temp (get_binres_type v1) in
                                 let btyp = get_binres_type v2  in
                                 let ptrsz = get_ptrsize_type btyp in
                                 let baddr = gl_atm v2 in
                                 gen_binres_type(List.tl btyp) @
                                 (incr_by_ptrsz v1 ptrsz off) @
                                 (add_base_offset btyp baddr off)
        | Addrof(v) -> let v1 = expr v in gen_addr_lst v1
        | Negof(v)  -> let v1 = expr v in
                        gen_binres_type( (get_binres_type v1))
                        @ v1 @ gen_atom (Neg(gl_atm v1))
        | Noexpr ->[Atom(Lit(0))]
        | Null ->  (gen_binres_type [Int]) @ gen_atom (Lit 0)
    in
let rec stmt = function
    Block sl ->
      (List.fold_left (fun str lst -> str @ lst) [] (List.map stmt sl) )
    | Expr e -> expr e
    | Return e ->
      let v1 = expr e in
        v1 @ [Rval (gl_atm v1)]
    | If (p, t, f) ->
      let pval = expr p and tval = stmt t and fval = stmt f in
        let v4 = (gl_atm pval) in
          let l1 = (get_ifelse_label 0) and l2 = (get_ifelse_label 1) in
            (match fval with
             [] -> pval @ [Predicate (v4,false, l2)] @ tval  @ [Label l2]
            | _ -> pval @ [Predicate (v4,false, l1)] @ tval  @ [Branch (l2)]
                      @ [Label l1] @ fval @ [Label l2])
    | For (asn, cmp, inc, b) ->
            stmt (Block (
                [Expr (asn); While(cmp, Block([b;Expr(inc)]))]
                ))

    | While (e, b) ->
      let v1 = stmt b and v2 = expr e and l0 = (get_loop_label 0)
        and l1 = (get_loop_label 1) in
        let v3 = (gl_atm v2) in
          [Branch l1] @ [Label l0] @ v1 @ [Label l1] @ v2 @ [Predicate
          (v3,true,l0)]
```

```
  | _ -> []
in

let stmtblock = (gen_vararr fdecl.locals) @ (stmt (Block fdecl.body)) in

(*[Global([Debug("Debug Message"); Debug("Yellow")])] @*)
[Fstart(fdecl.fname, (conv2_byt_lvar fdecl.formals), stmtblock, !curr_offset)]

in let env = { function_index = function_indexes;
                        global_index    = global_indexes;
               struct_index    = struct_indexes;
                        local_index     = StringMap.empty
             }
in

(* Code executed to start the program *)
let entry_function = try
  (StringMap.find "main" function_indexes); []
  with Not_found ->err ("no \"main\" function")
in
(* Compile the functions *)
List.concat (entry_function :: List.map (translate env) functions);;
(* TODO: Globals might need to be passed before at the point where
 * entry_function is present. Globals can be passed as a list, like that of
 * Fstart *)
```

8.4 CPI.ML

```
type action = Ast | Interpret | Bytecode | Compile


let usage_msg =
    "Cπ - Simplified C compiler for ARM V6\n" ^
    "cpi FILE [-o OUTFILE]\n"

(* Default argument values *)
let out_file = ref "out"
let use_stdin = ref false
let use_stdout = ref false
let create_binary = ref false
let debug_bytecode = ref false
let debug_sast = ref false
let no_sast = ref false

(* Command line args *)
```

```
let speclist =
    [
        ("--stdin", Arg.Set use_stdin, "\tRead from stdin" );
        ("--stdout", Arg.Set use_stdout, "\tOutput to stdout" );
        ("-b", Arg.Set debug_bytecode, "\t\tPrint out bytecode" );
        ("-sast", Arg.Set debug_sast, "\tPrint out sast" );
        ("--binary", Arg.Set create_binary,
        "\tCreate binary executable (only if -o is set)" );
        ("-o", Arg.String (fun x -> out_file := x), "\t\tSet output file");
        ("-tc", Arg.Set no_sast, "\t\tTurn off typechecking");
    ]


let save filename s =
    let channel = open_out filename in
    output_string channel s;
    close_out channel


(* Create and save executable binary file from assembly file *)
let create_binary_file filename =
    let filename_asm = filename ^ ".s" in
    let filename_obj = filename ^ ".o" in
    Sys.command ("as -o " ^ filename_obj ^ " " ^ filename_asm);
    Sys.command ("gcc -o " ^ filename ^ " " ^ filename_obj);
    (* Now clean up *)
    Sys.command ("rm -f " ^ filename_asm);
    Sys.command ("rm -f " ^ filename_obj);
    ()


let sast in_channel =
    let lexbuf = Lexing.from_channel in_channel in
    let ast = Parser.program Scanner.token lexbuf in
    Typecheck.type_check_prog ast


let program in_channel =
    let lexbuf = Lexing.from_channel in_channel in
    let ast = Parser.program Scanner.token lexbuf in
      Compile.translate ast

let program_tc in_channel =
    let lexbuf = Lexing.from_channel in_channel in
    let ast = Parser.program Scanner.token lexbuf in
      Typecheck.type_check_prog ast;
      Compile.translate ast
```

```
(* Compiles from an input channel (stdin or source file) *)
(* If --stdout flag set, then print to stdout. else, save to out_file *)
let compile in_channel out_file =
    let asm =
      if !no_sast then (Execute.execute_prog (program in_channel) )
      else  (Execute.execute_prog (program_tc in_channel) ) in
        if !use_stdout then print_string asm
        else
            save (out_file ^ ".s") asm;
            if !create_binary then create_binary_file out_file

let print_bytecode in_channel out_file =
    let bytecode = Debug.dbg_str_program (program in_channel)  in
        if !use_stdout then print_string bytecode
        else save (out_file ^ ".bytecode") bytecode

let print_sast in_channel out_file =
    let sast_str = Debug.dbg_str_sast (sast in_channel)  in
        if !use_stdout then print_string sast_str
        else save (out_file ^ ".sast") sast_str

(* MAIN *)
let main =
    (* Assume all anonymous arguments are source files and add them to
     * source_files list *)
    let source_files = ref [] in
        Arg.parse speclist (fun file -> source_files := file::!source_files ) usage_msg;

        (* If --stdin flag is set, read source from stdin *)
        (* Else, read from input source files *)
        if !use_stdin then (compile stdin !out_file) else
            List.iter (fun f -> compile (open_in f) !out_file ) !source_files;

        if !use_stdin && !debug_bytecode then (print_bytecode stdin !out_file)
        else if !debug_bytecode then
            List.iter (fun f -> print_bytecode (open_in f) !out_file ) !source_files;

        if !use_stdin && !debug_sast then (print_sast stdin !out_file)
        else if !debug_sast then
            List.iter (fun f -> print_sast (open_in f) !out_file ) !source_files;
```

8.5 DEBUG.ML

```
open Ast
open Sast
open Bytecode
```

```
let rec p tab_count = if (tab_count = 0 ) then "" else "\t" ^ p (tab_count-1);;

let dbg_str_of_typs typ = match typ with
                        Void -> "Void"
                        | Char -> "Char"
                        | Int -> "Int"
                        | Ptr -> "Ptr"
                        | Arr(sz) -> "Arr"
                        | Struct(sname) -> "Struct "
                        | Err -> "Error"

let dbg_typ ty =
  (List.fold_left (fun s t -> s ^ (dbg_str_of_typs t)) "" ty);;

let dbg_typ_ll ty =
  (List.fold_left (fun s t -> s ^ " " ^ (dbg_typ t)) "" ty);;

let rec dbg_str_Lvar lvar tabs = match lvar with
                 Lvar(off,sz) -> "Lvar Offset: " ^ string_of_int off ^
                              " Size: " ^ string_of_int sz
                | Lit (i) -> "Literal: " ^ string_of_int i
                | Cchar (ch) -> "Const char :" ^ String.make 1 ch
                | Sstr (str, label) -> "String: " ^ str ^ " Label: " ^ label
                | Gvar (_,_) -> "Globals: need implementation" (* Global var (name,size) *)
                | Pntr (atm, sz) ->
                   "Pointer: " ^
                   "\n" ^ p (tabs+2) ^ "Value | " ^ (dbg_str_Lvar atm (tabs+1)) ^
                   "\n" ^ p (tabs+2) ^ "Size  | " ^ (string_of_int sz)
                | Addr (atm)->
                   "Address: " ^
                   "\n" ^ p (tabs+2) ^ "Value | " ^ (dbg_str_Lvar atm (tabs+1))
                | Neg (atm)->
                   "\n" ^ p (tabs+2) ^ "Negative: \n" ^
                   "\n" ^ p (tabs+2) ^ "Value | " ^ (dbg_str_Lvar atm (tabs+1))
                | Debug(str) -> str

let dbg_str_print str = raise (Failure ("Debug msg: \n" ^str));;

let dbg_str_resolve r tabs = match r with
                | Dot  -> p (tabs) ^ "Dot(.)"
                | Ind -> p (tabs) ^ "Ind(->)"

let dbg_str_op o tabs = match o with
                | Add  -> p (tabs) ^ "Add"
                | Sub -> p (tabs) ^ "Sub"
                | Mult -> p (tabs) ^ "Mult"
                | Div -> p (tabs) ^ "Div"
```

```
                  | Equal -> p (tabs) ^ "Equal"
                  | Neq -> p (tabs) ^ "Neq"
                  | Less -> p (tabs) ^ "Less"
                  | Leq -> p (tabs) ^ "Leq"
                  | Greater -> p (tabs) ^ "Greater"
                  | Geq -> p (tabs) ^ "Geq";;

let dbg_str_bstmt bstm tabs = match bstm with
                    Atom (atm) -> p tabs ^ "Atom -> \n"
                      ^ p (tabs+1) ^ dbg_str_Lvar atm (tabs+1)
                  | BinEval  (dst, var1, op, var2) -> "BinEval -> \n"
                      ^ p (tabs+1) ^ "Dst   |" ^ (dbg_str_Lvar dst (tabs+1))^ "\n"
                      ^ p (tabs+1) ^ "Var1  |" ^ (dbg_str_Lvar var1 (tabs+1) ) ^ "\n"
                      ^ p (tabs+1) ^ "Op    |" ^ (dbg_str_op op 0)^ "\n"
                      ^ p (tabs+1) ^ "Var2  |" ^ (dbg_str_Lvar var2 (tabs+1))
                  | Fcall (fname, args,ret ) -> "Fcall -> \n"
                      ^ p (tabs+1) ^ "fname |" ^ fname ^ "\n"
                      ^ p (tabs+1) ^ "args  |" ^
                        (List.fold_left
                        (fun s t -> s ^ " " ^ (dbg_str_Lvar t (tabs+1))) "" args) ^"\n"
                      ^ p (tabs+1) ^ "ret   |" ^ (dbg_str_Lvar ret (tabs+1))
                  | Assgmt (dst, src) ->  "Assignment -> \n"
                      ^ p (tabs+1) ^ "dst   |" ^ (dbg_str_Lvar dst (tabs +1)) ^ "\n"
                      ^ p (tabs+1) ^ "src   |" ^ (dbg_str_Lvar src (tabs +1))
                  | Label (a)-> "Label -> \n"
                      ^ p (tabs+1) ^ a
                  | Predicate (pred, b,label )-> "Predicate -> \n"
                      ^ p (tabs+1) ^ "Pred  |" ^ (dbg_str_Lvar pred (tabs+1)) ^"\n"
                      ^ p (tabs+1) ^ "Label |" ^ label
                  | Branch(b)-> "Branch -> \n"
                      ^ p (tabs+1) ^ b
                  | BinRes(ty) -> "BinRes: -> \n"
                      ^ p (tabs+1) ^ (List.fold_left (fun s t -> s ^
                      (dbg_str_of_typs t)) "" ty)
                  | VarArr(_,_) ->"VarArr: -> \n" (*raise (Failure ("Unexpected:
                  VarArr")) *)
                  |Rval (rval) -> " Rval" ^ "\n"
                      ^ p (tabs+1) ^ "Rvalue |" ^ (dbg_str_Lvar rval (tabs+1)) ;;

let dbg_str_bstmlist lst fname sz = fname ^ " stack size = " ^ (string_of_int
sz) ^ "\n" ^
      (List.fold_left (fun s bstm -> s^"\n" ^ (dbg_str_bstmt bstm 0)) "" lst);;


let dbg_str_program prog =
      let rec dbg_str_proglst =
              function
              [] -> ""
```

```
             | hd :: tl ->
                (match hd with
                    Global (atmlst) -> "" (* dbg_print (List.hd atmlst) (*TODO: Global
                    functions code *)*)
                  | Fstart (fname, formals, body, stack_sz) ->
                    dbg_str_bstmlist body fname stack_sz
                ) ^ (dbg_str_proglst tl)
    in dbg_str_proglst prog;;

let rec dbg_str_sast_expr sast_expr tabs = match sast_expr with
  | Literal_t(i, t) ->
      dbg_typ t
  | String_t(s, t) ->
      dbg_typ t
  | Addrof_t(e, t) ->
      p (tabs) ^ dbg_typ t
    ^ p (tabs+1) ^ "&"
    ^ dbg_str_sast_expr e (tabs+1)   ^ "\n"
  | Negof_t(e, t) ->
      p (tabs) ^ "-("
    ^ dbg_str_sast_expr e (tabs+1)  ^ ")\n"
  | ConstCh_t(s, t) ->
       dbg_typ t
  | Id_t(s, t) ->
       dbg_typ t
  | MultiId_t(e1, r, e2, t) ->
      p (tabs) ^ "MultiId" ^ "\n"
    ^ p (tabs) ^ dbg_str_sast_expr e1 (tabs+1) ^ "\n"
    ^ p (tabs) ^ dbg_str_resolve r  (tabs +1) ^ "\n"
    ^ p (tabs) ^ dbg_str_sast_expr e2 (tabs+1)  ^ "\n"
  | Pointer_t(e, t) ->
      p (tabs) ^ dbg_typ t ^ "\n"
    ^ p (tabs+1) ^ "*"
    ^ dbg_str_sast_expr e (tabs+1) ^ "\n"
  | Array_t(e1, e2, t) ->
      p (tabs) ^ dbg_typ t ^ "("
    ^ dbg_str_sast_expr e1 (0) ^ "[" ^ dbg_str_sast_expr e2 (0) ^ "])"
  | Binop_t(e1, o, e2, t) ->
      p (tabs) ^ dbg_typ t ^ "\n"
    ^ p (tabs+1) ^ dbg_str_sast_expr e1 (0) ^ " "
    ^ dbg_str_op o (0)  ^ " "
    ^ dbg_str_sast_expr e2 (0) ^ "\n"
  | Assign_t(e1, e2, t) ->
      p (tabs) ^ dbg_typ t ^ "\n"
    ^ p (tabs+1) ^ dbg_str_sast_expr e1 (0) ^ " = "
    ^ dbg_str_sast_expr e2 (0) ^ "\n"
  | Call_t(s, e_l, t) ->
      p (tabs) ^ dbg_typ t ^ "\n"
```

```
      ^ p (tabs+1) ^ s ^ "( "
      ^ (List.fold_left
        (fun s e -> s ^(dbg_str_sast_expr e (1))) "" e_l) ^ ")\n"
    | Noexpr_t(t) ->
        p (tabs) ^ "No Expression" ^ "\n"
    | Null_t(t) ->
        p (tabs) ^(dbg_typ t) ^ "\n";;
(*
let rec dbg_str_sast_expr sast_expr tabs = match sast_expr with
    | Literal_t(i, t) ->
        p (tabs) ^ "Literal:" ^ string_of_int i
    | String_t(s, t) ->
        p (tabs) ^ "String: " ^ s
    | Addrof_t(e, t) ->
        p (tabs) ^ "Addrof:" ^ "\n"
      ^ p (tabs) ^ dbg_str_sast_expr e (tabs+1)   ^ "\n"
    | Negof_t(e, t) ->
        p (tabs) ^ "Neg:"
      ^ p (tabs) ^ dbg_str_sast_expr e (tabs+1)  ^ "\n"
    | ConstCh_t(s, t) ->
        p (tabs) ^ "Char: " ^ s
    | Id_t(s, t) ->
        p (tabs) ^ "Id: " ^ s
    | MultiId_t(e1, r, e2, t) ->
        p (tabs) ^ "MultiId" ^ "\n"
      ^ p (tabs) ^ dbg_str_sast_expr e1 (tabs+1) ^ "\n"
      ^ p (tabs) ^ dbg_str_resolve r  (tabs +1) ^ "\n"
      ^ p (tabs) ^ dbg_str_sast_expr e2 (tabs+1)  ^ "\n"
    | Pointer_t(e, t) ->
        p (tabs) ^ "Pointer:" ^ "\n"
      ^ p (tabs) ^ dbg_str_sast_expr e (tabs+1) ^ "\n"
    | Array_t(s, e, t) ->
        p (tabs) ^ "Array: " ^ s ^ "\n"
      ^ p (tabs) ^ dbg_str_sast_expr e (tabs+1) ^ "\n"
    | Binop_t(e1, o, e2, t) ->
        p (tabs) ^ "Binop:" ^ "\n"
      ^ p (tabs) ^ dbg_str_sast_expr e1 (tabs+1) ^ "\n"
      ^ p (tabs) ^ dbg_str_op o (tabs+1) ^ "\n"
      ^ p (tabs) ^ dbg_str_sast_expr e2 (tabs+1) ^ "\n"
    | Assign_t(e1, e2, t) ->
        p (tabs) ^ "Assign: " ^ "\n"
      ^ p (tabs) ^ dbg_str_sast_expr e1 (tabs+1) ^ "\n"
      ^ p (tabs) ^ dbg_str_sast_expr e2 (tabs+1) ^ "\n"
    | Call_t(s, e_l, t) ->
        p (tabs) ^ "Call: " ^ "\n"
      ^ p (tabs) ^ (List.fold_left
        (fun s e -> s ^(dbg_str_sast_expr e (tabs+1))) "" e_l) ^ "\n"
    | Noexpr_t ->
```

```
    p (tabs) ^ "No Expression" ^ "\n";;
*)
let rec dbg_str_sast_stmt sast_stm tabs = match sast_stm with
    Block_t(stmlst) -> "Block -> "
    ^ (List.fold_left (fun s sast_stm -> s^"\n" ^ (dbg_str_sast_stmt sast_stm
      (tabs+1))) "" stmlst)
  | Expr_t(e) -> "Expr -> \n"
    ^ (dbg_str_sast_expr e (tabs+1))
  | Return_t(e) -> "Return -> \n"
    ^ (dbg_str_sast_expr e (tabs+1))
  | If_t(e, t_s, f_s) -> "If -> \n"
    ^ "Predicate Expr:\n" ^ (dbg_str_sast_expr e (tabs+1))
    ^ "True Stmt:\n" ^ (dbg_str_sast_stmt t_s (tabs+1))
    ^ "False Stmt:\n" ^ (dbg_str_sast_stmt f_s (tabs+1))
  | For_t(asn, cond, inc, s) -> "For -> \n"
    ^ "Assingment Expr:\n" ^ (dbg_str_sast_expr asn (tabs+1))
    ^ "Conditional Expr:\n" ^ (dbg_str_sast_expr cond (tabs+1))
    ^ "Increment Expr:\n" ^ (dbg_str_sast_expr inc (tabs+1))
    ^ "For Stmt:\n" ^ (dbg_str_sast_stmt s (tabs+1))
  | While_t(e, s) -> "While -> \n"
    ^ "While Expr:\n" ^ (dbg_str_sast_expr e (tabs+1))
    ^ "While Stmt:\n" ^ (dbg_str_sast_stmt s (tabs+1))


let dbg_str_sast_stmlist lst name tabs = name ^
      (List.fold_left (fun s sast_stm -> s^"\n" ^ (dbg_str_sast_stmt sast_stm
      tabs)) "" lst);;

let dbg_str_sast sast =
  let get_sast_lst(prog, s) = s in
  let sast_lst = get_sast_lst(sast) in
  let rec dbg_str_sastlst =
    function
      [] -> ""
    | hd :: tl ->
      (match hd with
      Sast (fname, formals, body) ->
        dbg_str_sast_stmlist body ("Function: " ^ fname ^ "\n") 0)
      ^ (dbg_str_sastlst tl)
    in dbg_str_sastlst sast_lst
```

8.6 EXECUTE.ML

```
open Ast
open Bytecode
```

```
module IntMap = Map.Make(
struct type t = int
let compare = compare end
)

module StringMap = Map.Make(String)

type byc_gvar_entry = { (*TODO: add more require elements*)
        label: string;
}

type byc_env = {
        global_index: byc_gvar_entry StringMap.t;
}


let execute_prog program =
        let p asm = "\t " ^ asm ^ "\n"
        and size_stmfd = 4 (* Total size pushed using stmfd -4 *)
        and align_size = 4
        in
let get_aligned_sz sz =
(align_size * int_of_float(ceil ((float_of_int sz ) /.
                        (float_of_int align_size))))
        in
let dbg_print var = match var with
        Lvar(off,sz) -> "Offset: " ^ string_of_int off ^
                        "Size: " ^ (string_of_int sz) ^ "\n"
        | Sstr(s, l) -> "String: " ^ s ^ "Label: " ^ l ^ "\n"
        | Debug(s) -> "Debug: " ^ s ^"\n"
        | _ -> "IMPLEMENT"
        in
let dbg_raise_error_atom str a = raise(Failure( str ^
                (match a with
                  Lit (i) -> "Literal " ^ string_of_int i
                | Cchar(ch) -> "Const Char"
                | Sstr (s, l) -> "StringConst "^s
                | Lvar (o,s) -> " Lvar"
                | Gvar (_,_) -> "Gvar"
                | Pntr (_,_) -> "Pntr"
                | Addr (_) -> "Addr"
                | Debug (_)  -> "Debug"
                | Neg (_) -> "Negative")))
        in
let size_of_lvar l = match l with
                Lvar(off,sz)-> sz
                   | Gvar(n,s)-> s
```

```
                  | _ -> raise (Failure("Cannot generate size"))
        in
let idx_to_offset off = off + size_stmfd
        in
let rec print_atom_lst  = function
      [] -> ""
    | hd :: tl ->
        dbg_print hd ^ (print_atom_lst tl)
        in
let function_code_gen fname formals body stack_sz =
        let branch lb = p ("b " ^ lb) in
        let gen_label lbl = lbl ^ ":" ^ "\n" in
        let exit_label = fname ^ "_exit" in

        (* Note register r4 will be left as a temporary register
         * so that anybody can use .eg in gen_ldr_str_code *)
        let rec gen_ldr_str_code oper sym reg atm =
                let pre sz = if sz != 0 then(  oper ^ (if sz = 1 then "b" else "")
                        ^" "^ reg ^", ") else "" in
        match atm with
          Lit (i) -> p ( (pre 4)  ^ sym ^ string_of_int i)
        | Cchar (ch) -> p ((pre 1) ^ sym ^ string_of_int (int_of_char ch))
        | Lvar (off, sz) -> if sz = 0 then "" else ( p ( (pre sz) ^ "[fp,#-" ^ string_of_int
                                (idx_to_offset off) ^"]"))
        | Gvar (vname, sz) -> "" (*TODO *)
        | Neg  (vnm) -> p ("rsb " ^reg^ ", " ^reg ^", #0")
        | Addr (vnm) -> (match vnm with
                  Lvar(off,sz) -> (if sz=0 then "" else
                        p ("sub " ^reg^", fp,#" ^
                        string_of_int (idx_to_offset off)))
                | Gvar(vname,sz) -> "" (*TODO: Globals*)
                | Pntr(dst,psz) -> gen_ldr_str_code oper sym reg dst
                | _ as l -> dbg_raise_error_atom "Addr: " l)
        | Pntr (dst,psz) -> (match dst with
                  Lvar(off,sz) -> (if sz=0 then ""
                        else (gen_ldr_str_code "ldr" "=" "r4" dst) ^
                        p ((pre psz) ^ "[r4,#0]"))
                | Pntr(_,_) -> (gen_ldr_str_code "ldr" "=" "r4" dst) ^
                                p((pre psz) ^ "[r4,#0]")
                | Gvar(vname,sz) -> "" (*TODO: Globals*)
                | _ as l -> dbg_raise_error_atom "Pntr: " l
                )
        | Sstr (s, l) -> p ( "ldr r0, =" ^ l)
        | Debug (s) -> s
       in
       let load_code reg var = (* Load variable var to register reg *)
                gen_ldr_str_code "ldr" "=" reg var
       and store_code reg var =
```

```
                gen_ldr_str_code "str" "#" reg var in
let incr_stack sz =
p ("sub sp, sp," ^ sz )
  in
let bin_eval dst var1 op var2 =
        let oper = (match op with
        Add -> p "adds r3, r0, r1"
      | Sub -> p "subs r3, r0, r1"
      | Mult ->p "muls r3, r0, r1"
      | Div -> p "bl __aeabi_idiv" ^
               p "mov r3, r0"
      | Equal ->
               p "cmp r0, r1" ^
               p "moveq r3,#1" ^
               p "movne r3,#0" ^
               p "uxtb r3,r3"(*TODO-check the need*)
      | Neq ->
               p "cmp r0, r1" ^
               p "moveq r3,#0" ^
               p "movne r3,#1" ^
               p "uxtb r3,r3"
      | Less ->
               p "cmp r0, r1" ^
               p "movlt r3,#1" ^
               p "movge r3,#0" ^
               p "uxtb r3,r3"
      | Leq ->
               p "cmp r0, r1" ^
               p "movle r3,#1" ^
               p "movgt r3,#0"^
               p "uxtb r3,r3"
      | Greater ->
               p "cmp r0, r1"^
               p "movgt r3,#1"^
               p "movle r3,#0"^
               p "uxtb r3,r3"
      | Geq ->
               p "cmp r0, r1"^
               p "movge r3,#1"^
               p "movlt r3,#0"^
               p "uxtb r3,r3"
        )
        in
(load_code "r0" var1) ^ (load_code "r1" var2) ^ oper ^ (store_code "r3" dst)
        in
let function_call fname args ret=
  let rec fcall i = function
    [] -> ""
```

```
    | hd :: tl -> (load_code ("r" ^ string_of_int i) hd ) ^ (fcall (i+1) tl) in
      fcall 0 args ^ ("\n\t bl  " ^ fname ^ "\n" ) ^ (store_code "r0" ret)
    (* TODO implement properly *)
        in
let predicate cond jmpontrue label =
        let brn = if jmpontrue then "\t bne " else "\t beq " in
        (load_code "r0" cond) ^ "\t cmp r0,#0\n" ^ brn ^ label ^ "\n"
        in
let var_array ptr sz =
  (* Code for alignment of sz *)
        let align_bits = align_size / 2 in
        (load_code "r0" sz) ^
        p ("lsr r1,r0,#"^ (string_of_int align_bits)) ^
        p ("lsl r1,r1,#"^ (string_of_int align_bits)) ^
        p ("cmp r1,r0") ^
        p ("movne r0,#" ^ (string_of_int align_size) ) ^
        p ("moveq r0,#0" ) ^
        p ("add r3,r0,r1") ^
        (incr_stack "r3") ^
        p ("mov r0,sp") ^
        store_code "r0" ptr
        in
let asm_code_gen = function
    Atom (atm) -> ""
  | BinEval  (dst, var1, op, var2) -> bin_eval dst var1 op var2
  | Assgmt (dst, src) -> (load_code "r0" src) ^ (store_code "r0" dst)
  | Fcall (fname, args,ret) ->  function_call fname args ret
  | Rval var -> (load_code "r0" var) ^ (branch exit_label)
  | Branch label -> branch label
  | Label label -> gen_label label
  | Predicate (cond,jmpontrue,label) -> predicate cond jmpontrue label
  | BinRes (_) -> ""
  | VarArr(ptr,sz) -> var_array ptr sz
in
let non_atom lst = (List.filter (fun ele -> match ele with
                 Atom (atm ) -> false
               | BinRes(_) -> false
               | _ -> true) lst)
in
let mem_code_gen = function
    Atom (atm) ->
     ( match atm with
        Sstr (s, l) ->  l ^ ":\n" ^
                      p (".asciz   " ^  s)
       | _ -> "" )
  | _ -> ""
in
let func_start_code =
```

```
        ".data\n" ^
        (List.fold_left
                (fun str lst -> str ^ (mem_code_gen lst))
                "" body) ^ "\n" ^
        ".text\n" ^
        (* Code generation for function *)
        ".global " ^ fname ^ "\n" ^
            fname ^ ":\n" ^
                   (p "stmfd sp!, {fp, lr}") ^
                   p ("add fp, sp,#"^ string_of_int size_stmfd)  ^
                  (* List.fold_left (fun s v->s ^ "\n" ^ (dbg_print v)) "" temps
                   ^*)
                   (incr_stack ("#" ^ (string_of_int stack_sz)))^
                   let rec formals_push_code i = if i < 0 then "" else
                            (formals_push_code (i-1)) ^
                            (store_code ("r" ^ string_of_int i) (List.nth formals i))
                    in formals_push_code ((List.length formals) -1)
                    (* TODO : if the variable size is 1 byte, strb should be
                     * used instead and the var_size should be updated
                     * accordingly *)
        and func_end_code = (gen_label exit_label) ^
                p "sub sp, fp, #4" ^
                       p "ldmfd sp!, {fp, pc}" ^ "\n"
        in func_start_code ^
        (List.fold_left
                (fun str lst -> str ^ (asm_code_gen lst))
                "" (non_atom body))
        ^ func_end_code
in
let env = {
        global_index = StringMap.empty;
         }
in let rec print_program =
        function
        [] -> ""
        | hd :: tl ->
           (match hd with
               Global (atmlst) -> print_atom_lst atmlst (* dbg_print (List.hd atmlst) (*TODO:
Global
               functions code *)*)
             | Fstart (fname, formals, body, stack_sz) ->
               (function_code_gen fname formals body
                 (get_aligned_sz stack_sz)
               )
           )
                          ^ (print_program tl)
in (print_program program)
```

8.7 PARSER.MLY

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA LSUBS RSUBS
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE INT CHAR STRUCT VOID
%token AMPERSAND INDIRECTION DOT
%token LAND LOR
%token <string> CONSTCHAR
%token <string> STRING
%token <string> ID
%token <int> LITERAL
%token NULL
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left LOR
%left LAND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%left INDIRECTION DOT LPAREN RPAREN LSUBS RSUBS

%start program
%type <Ast.program> program

%%

program:
        /* nothing */ { {gdecls=[];sdecls=[];fdecls=[] } }
 | program sdecl { {gdecls= $1.gdecls; sdecls=$2::$1.sdecls; fdecls= $1.fdecls} }
 | program vdecl { {gdecls= $2::$1.gdecls; sdecls=$1.sdecls; fdecls=$1.fdecls} }
 | program fdecl { {gdecls= $1.gdecls; sdecls=$1.sdecls; fdecls=$2::$1.fdecls} }


fdecl:
   retval formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { { fname = snd $1;
         formals = $2;
```

```
          locals = List.rev $5;
          body = List.rev $6;
          ret = fst $1
          } }

retval:
        INT ID LPAREN { [Int], $2  }
        |CHAR ID LPAREN { [Char], $2  }
        |VOID ID LPAREN { [Void], $2  }

sdecl:
        STRUCT ID LBRACE vdecl_list RBRACE SEMI
        { {
          sname = $2;
          smembers = $4;
           }
        }

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

formal_list:
    tdecl                   { [$1] }
  | formal_list COMMA tdecl {
                  (match List.hd $3.vtype with
                  Arr(s) -> (match s with
                    Id(id) -> raise( Failure("Array declaration: "^
                      "variable not allowed in" ^
                      "funciton argument"))
                    |_ -> $3)
                  | _ -> $3) :: $1



    }

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

vdecl:
   | tdecl SEMI { match List.hd $1.vtype with
                  Arr(s) -> (match s with
                    Noexpr -> raise( Failure("Array declaration: Size not specified"))
                    |_ -> $1)
                  | _ -> $1
                }
```

```
tdecl:
       INT rdecl      {
                        {
                        vname = $2.vname;
                        vtype = $2.vtype @ [Int]
                        }
                       }
     | CHAR rdecl      {
                        {
                        vname = $2.vname;
                        vtype = $2.vtype @ [Char]
                        }
                       }
     | STRUCT ID rdecl {
                      { vname = $3.vname;
                        vtype = $3.vtype @ [Struct($2)]
                      }
                       }

rdecl:
        ID            {
                      { vname = $1;
                        vtype = []
                      }
                     }
        | arrdecl      { $1 }
        | TIMES rdecl   { {
                        vname = $2.vname;
                        vtype = $2.vtype @ [Ptr];
                        } }

arrdecl:
        ID LSUBS LITERAL RSUBS { {
          vname = $1;
          vtype = [Arr(Literal($3))]
           } }
        | ID LSUBS RSUBS { {
          vname = $1;
          vtype = [Arr(Noexpr)]
           } }
        | ID LSUBS ID RSUBS { {
          vname = $1;
          vtype = [Arr(Id($3))]
           } }

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

```
stmt:
    expr SEMI { Expr($1) }
  | RETURN expr SEMI { Return($2) }
  | RETURN SEMI { Return(Noexpr) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }

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

expr:
    LITERAL          { Literal($1) }
  | NULL             { Null }
  | MINUS LITERAL    { Literal(-$2) }
  | PLUS LITERAL     { Literal($2) }
  | AMPERSAND lvalue { Addrof($2)  }
  | MINUS lvalue     { Negof($2)   }
  | PLUS lvalue      { $2 }
  | CONSTCHAR        { ConstCh($1) }
  | STRING           { String($1) }
  | expr PLUS   expr { Binop($1, Add,   $3) }
  | expr MINUS  expr { Binop($1, Sub,   $3) }
  | expr TIMES  expr { Binop($1, Mult,  $3) }
  | expr DIVIDE expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr LOR    expr { Binop($1, Lor,   $3) }
  | expr LAND   expr { Binop($1, Land,   $3) }
  | expr ASSIGN expr   { Assign($1, $3) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | expr DOT var { MultiId($1,Dot,$3) }
  | expr INDIRECTION var { MultiId($1,Ind,$3) }
  | lvalue           { $1 }

lvalue:
        ptr   {$1}
        |var  {$1}
        |LPAREN expr RPAREN {$2}
```

```
ptr:
        TIMES expr {Pointer($2)}

var:
        ID      { Id($1) }
        | arr   { Array( fst $1, snd $1) }

arr:
        ID LSUBS expr RSUBS { Id($1),$3 }

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

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

8.8 SAST.ML

```
open Ast

module StringMap = Map.Make(String)

type expr_t =
  | Literal_t of int * cpitypes list
  | String_t of string * cpitypes list
  | Addrof_t of expr_t * cpitypes list
  | Negof_t of expr_t * cpitypes list
  | ConstCh_t of string * cpitypes list
  | Id_t of string * cpitypes list
  | MultiId_t of expr_t * resolve * expr_t * cpitypes list
  | Pointer_t of expr_t * cpitypes list
  | Array_t of expr_t * expr_t * cpitypes list
  | Binop_t of expr_t * op * expr_t * cpitypes list
  | Assign_t of expr_t * expr_t * cpitypes list
  | Call_t of string * expr_t list * cpitypes list
  | Noexpr_t of cpitypes list
  | Null_t of cpitypes list

type stmt_t =
    Block_t of stmt_t list
  | Expr_t of expr_t
  | Return_t of expr_t
```

```
  | If_t of expr_t * stmt_t * stmt_t
  | For_t of expr_t * expr_t * expr_t * stmt_t
  | While_t of expr_t * stmt_t

type prog_t =
  Sast of string * expr_t list * stmt_t list

(*
type prog_t = {
  fname: string;
  formals: expr_t list;
  stmts: stmt_t list;
  prog: Ast.program
}
*)
type var_entry = {
  offset:int;
  typ: cpitypes list
}

type func_entry = {
  param : var_entry list;
  ret_ty : cpitypes list
}

type struct_entry = {
  size: int;
  memb_index: var_entry StringMap.t
}

(* Symbol table: Information about all the names in scope *)
type envt = {
  function_index : func_entry StringMap.t; (* Index for each function *)
  struct_index   : struct_entry StringMap.t;
  global_index   : var_entry StringMap.t; (* "Address" for global variables *)
  local_index    : var_entry StringMap.t; (* FP offset for args, locals *)
}
```

8.9 SCANNER.MLL

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| "//"     { line_comment lexbuf }
| "#include"  { includes  lexbuf }
```

```
| '('       { LPAREN }
| ')'       { RPAREN }
| '{'       { LBRACE }
| '}'       { RBRACE }
| '['       { LSUBS }
| ']'       { RSUBS }
| ';'       { SEMI }
| ','       { COMMA }
| '+'       { PLUS }
| '-'       { MINUS }
| '*'       { TIMES }
| '&'       { AMPERSAND }
| '/'       { DIVIDE }
| '.'       { DOT }
| '='       { ASSIGN }
| "=="      { EQ }
| "!="      { NEQ }
| '<'       { LT }
| "<="      { LEQ }
| ">"       { GT }
| ">="      { GEQ }
| "->"      { INDIRECTION }
| "&&"      { LAND }
| "||"      { LOR }
| "if"      { IF }
| "else"    { ELSE }
| "for"     { FOR }
| "while"   { WHILE }
| "return"  { RETURN }
| "int"     { INT }
| "char"    { CHAR }
| "struct"  { STRUCT }
| "NULL"    { NULL }
| "void"    { VOID }
| ''' [ ^'''] as ch ''' { CONSTCHAR(ch) }
| '"' [^'"']* '"'  as lxm { STRING(lxm) }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }

and line_comment = parse
  "\n"  { token lexbuf }
| _     { line_comment lexbuf }
```

```
and includes = parse
  "\n"  { token lexbuf }
| _     { includes lexbuf }
```

8.10 TYPECHECK.ML

```
open Ast
open Sast
open Debug

module StringMap = Map.Make(String)

let rec get_size_type sindex = function
|[] ->   raise Exit
| hd::tl ->
  (match hd with
    Void -> 0
  | Char -> 1
  | Int
  | Ptr -> 4
  | Arr(sz) ->  (match sz with
        Literal(i) -> i
        | Id(id) -> get_size_type sindex [Ptr]
        | _ -> raise(Failure("lit_to_num: unexpected"))) * (get_size_type sindex tl)
  | Struct(sname) -> (StringMap.find sname sindex).size
  | _ -> raise (Failure ("Requesting size of wrong type")));;

let rec build_local_idx map sidx offset ?(rev =0) = (function
    [] -> map
  | hd:: tl ->
      offset := 0;
      build_local_idx ~rev:rev
    (if StringMap.mem hd.vname map then
      raise (Failure("Double declaration of " ^ hd.vname ))
     else
      StringMap.add hd.vname
        {
          offset = 0;
          typ = hd.vtype
        } map
      )
      sidx offset tl);;

let build_global_idx map = StringMap.empty;;

(* Translate a program in AST form into a bytecode program.  Throw an
```

```
 *   exception if something is wrong, e.g., a reference to an unknown
 *   variable or function *)
let type_check_prog prog =
  let structs = prog.sdecls
  and globals = prog.gdecls
  and functions = prog.fdecls in

(* Allocate "addresses" for each global variable *)
(* TODO Code generation for globals *)
let global_indexes = build_global_idx globals in

let struct_indexes = List.fold_left
  (fun map stct ->
    let soffset = ref 0 in
      let index = build_local_idx ~rev:1
        StringMap.empty map soffset (List.rev stct.smembers) in
        (
          StringMap.add stct.sname
            {
              size = !soffset;
              memb_index = index
            } map
        )
  )
  StringMap.empty structs
in

let f_index = List.fold_left
  (fun map fdecl ->
    let rec var_to_lst ind = function
      [] -> []
    (*TODO Check correct values*)
    | hd :: tl -> ( {offset =0; typ = hd.vtype} :: (var_to_lst (ind+1) tl)) in
      StringMap.add fdecl.fname
      {
        param = (var_to_lst 0 fdecl.formals);
        ret_ty = fdecl.ret
      }
      map
  )
  StringMap.empty functions
in

let f2_index =
  StringMap.add "printf"
  {
    param = [];
    ret_ty = [Int]
```

```
  }
  f_index
in

let f3_index =
  StringMap.add "scanf"
  {
    param = [];
    ret_ty = [Int]
  }
  f2_index
in

let f4_index =
  StringMap.add "malloc"
  {
    param = [];
    ret_ty = [Ptr;Void]
  }
  f3_index
in

let function_indexes =
  StringMap.add "free"
  {
    param = [];
    ret_ty = [Void]
  }
  f4_index
in

(* Translate a function in AST form into a list of bytecode statements *)
let type_check_func env fdecl=
 let curr_offset = ref 0 in
  let env =
    {
      env with local_index =
        (build_local_idx StringMap.empty env.struct_index curr_offset
        (fdecl.locals @ fdecl.formals))
    }
    in
  let rec conv2_expr_t = function
      [] -> []
    | hd::tl -> Id_t(hd.vname, hd.vtype) :: (conv2_expr_t tl)
    in
  let get_func_entry name =
    try StringMap.find name env.function_index
    with Not_found -> raise (Failure("Function not found: " ^ name))
```

```
  in
let get_func_decl_typs name =
  let param = (get_func_entry name).param in
    let rec conv_param2_typ_lst = function
      [] -> []
    | hd::tl -> hd.typ :: (conv_param2_typ_lst tl) in
    conv_param2_typ_lst param
  in
let get_type_varname table varname =
  try (StringMap.find varname table).typ
  with Not_found -> raise (Failure("Type Checking Varname not found: " ^ varname))
  in
let get_type_lst_expr_t = function
  | Literal_t(i, t) -> t
  | String_t(s, t) -> t
  | Addrof_t(e, t) -> t
  | Negof_t(e, t) -> t
  | ConstCh_t(s, t) -> t
  | Id_t(s, t) -> t
  | MultiId_t(e1, r, e2, t) -> t
  | Pointer_t(e, t) -> t
  | Array_t(s, e, t) -> t
  | Binop_t(e1, o, e2, t) -> t
  | Assign_t(e1, e2, t) -> t
  | Call_t(s, e_l, t) -> t
  | Noexpr_t(t) -> t
  | Null_t(t) -> t
  in
let is_arr typ_lst =
  match (List.hd typ_lst) with
  | Arr(_) -> true
  | _ -> false
  in
let get_typs_from_expr_t_lst param =
    let rec conv_el2_typ_lst = function
      [] -> []
    | hd::tl -> get_type_lst_expr_t hd :: (conv_el2_typ_lst tl) in
    conv_el2_typ_lst param
  in
let get_struct_table2 stct =
  (try (StringMap.find stct env.struct_index).memb_index
   with Not_found -> raise(Failure(" struct " ^ stct ^ " is not a type")))
  in
let get_struct_table typ_lst =
  match typ_lst with
  | [Struct(s)] -> (get_struct_table2 s)
  | [Ptr; Struct(s)] -> (get_struct_table2 s)
  | _ -> raise (Failure
```

```
        ("Variable is " ^ (dbg_typ typ_lst) ^ " and not a Struct"))
      in
    let rec lst_match list1 list2 =
      let typ_equal t1 t2 =
        if t1 = t2 then true else
        match t1, t2 with
        | Ptr, Arr(_) -> true
        | Arr(_), Ptr -> true
        | Arr(_), Arr(_) -> true
        | _ , _  -> false in
          match list1, list2 with
          | h1::t1, h2::t2 -> typ_equal h1 h2 && lst_match t1 t2
          | [_], _ -> false
          | _, [_] -> false
          | _, _ -> true
      in
    let is_int_or_char ty =
      if lst_match ty [Int] then true
        else if lst_match ty [Char] then true
        else false
      in
    let rec binop_result_type ?(strict=false) ty1 op ty2 =
          match ty1, ty2, op, strict with
          | [Int],  [Int],  _, _ -> [Int]
          | [Char], [Char], _, _ -> [Char]
          | _, _, _, true -> [Err]
          | [Int],  [Char], _, _ -> [Int]
          | [Char], [Int],  _, _ -> [Int]
          | Ptr::tl, [Int], Add, _ -> ty1
          | Ptr::tl, [Char], Add, _ -> ty1
          | Ptr::tl, [Int], Sub, _ -> ty1
          | Ptr::tl, [Char], Sub, _ -> ty1
          | [Int], Ptr::tl, Add, _ -> ty2
          | [Char], Ptr::tl, Add, _ -> ty2
          | [Int], Ptr::tl, Sub, _ -> ty2
          | [Char], Ptr::tl, Sub, _ -> ty2
          | Arr(s)::tl, [Int], Add, _ -> ty1
          | Arr(s)::tl, [Char], Add, _ -> ty1
          | Arr(s)::tl, [Int], Sub, _ -> ty1
          | Arr(s)::tl, [Char], Sub, _ -> ty1
          | [Int], Arr(s)::tl, Add, _ -> ty2
          | [Char], Arr(s)::tl, Add, _ -> ty2
          | [Int], Arr(s)::tl, Sub, _ -> ty2
          | [Char], Arr(s)::tl, Sub, _ -> ty2
  (*        | Ptr::t1, [Int], Equal, _ -> ty1
          | Ptr::t1, [Char], Equal, _ -> ty1 *)
          | Ptr::t1, [Ptr;Void], Equal, _ -> [Int]
          | Ptr::t1, [Ptr;Void], Neq, _ -> [Int]
```

```
      | Ptr::t1, Ptr::t2, Equal, _ -> binop_result_type ~strict:true t1 op t2
      | Ptr::t1, Ptr::t2, Neq, _ -> binop_result_type ~strict:true t1 op t2
      | Arr(s1)::t1, Arr(s2)::t2, Equal, _ -> binop_result_type ~strict:true t1 op t2
      | Arr(s1)::t1, Arr(s2)::t2, Neq, _ -> binop_result_type ~strict:true t1 op t2
      | _ , _ , _, _ -> [Err]
    in
  let assign_expr_result_type lh ty1 rh ty2 =
    let is_lh_arr t =
      (match (List.hd t) with
      | Arr(_) -> raise(Failure("Assign Type Error: Left hand side cannot be an
        array pointer"))
      | _ -> false)  in
    let is_lh_addr lh =
      (match lh with
      | Addrof_t(_,_) -> raise(Failure("Assign Type Error: Left hand side cannot be an
       address expression"))
      | _ -> false)  in
    if lst_match ty1 ty2 && not(is_lh_arr ty1) && not(is_lh_addr lh) then ty1
    else
       match ty1, ty2 with
        | [Int],  [Char] -> [Int]
        | [Char], [Int] -> [Char]
        | Ptr::t1, [Ptr;Void] -> ty1
        | _ , _  -> [Err]
    in
  let assign_result_type ty1 ty2 =
    if lst_match ty1 ty2 then ty1
    else
       match ty1, ty2 with
        | [Int],  [Char] -> [Int]
        | [Char], [Int] -> [Char]
        | _ , _  -> [Err]
    in
  let rec cmp_param_typ list1 list2 fname =
    (* Since printf and scanf are externally declared ignore them *)
    if (fname = "printf" || fname = "scanf" || fname="malloc" || fname="free") then true
    else
      match list1, list2 with
      | h1::t1, h2::t2 ->
        if (lst_match (assign_result_type h1 h2) [Err]) then
        false else cmp_param_typ t1 t2 fname
      | [_], _ -> false
      | _, [_] -> false
      | _, _ -> true
    in
let rec tc_expr ?(table = env.local_index) ?(strict=0) = function
    Literal i -> Literal_t(i, [Int])
  | String s -> String_t(s, [Ptr; Char])
```

```
| ConstCh(ch) -> ConstCh_t(ch, [Char])
| Id s ->
  let typ = get_type_varname table s in
  (*if is_arr typ then
    Id_t (s, [Ptr] @ (List.tl typ))
    else*) Id_t(s, typ)
| MultiId(fexpr,resolve,e) ->
  let v1 = tc_expr fexpr in
    let v1_type = get_type_lst_expr_t(v1) in
    (*let tab = (match v1_type with
      | [Struct(s)] -> get_struct_table s
      | [Prt;Struct(s)] -> get_struct_table s
      | _ -> raise(Failure("Variable is "^ (dbg_typ v1_type) ^" and not a
      Struct"))) in *)
    let tab = (get_struct_table v1_type) in
    let v2 = tc_expr ~table:tab ~strict:1 e in
    let v2_type = get_type_lst_expr_t(v2) in
      (*  raise (Failure ("Struct:
         First part is " ^ (dbg_typ v1_type) ^ " second part is " ^ (dbg_typ
         v2_type
         )) *)
    (match v1_type, resolve with
      | [Struct(s)], Dot -> MultiId_t(v1, Dot, v2, v2_type)
      | [Ptr;Struct(s)], Dot ->
        raise (Failure ("Struct Mismatch:
         Cannot use resolve operator " ^ (dbg_str_resolve resolve 0) ^ " with
         Struct "^ s ^ " which has type " ^ (dbg_typ v1_type)))
      | [Struct(s)], Ind ->
        raise (Failure ("Struct Mismatch:
         Cannot use resolve operator " ^ (dbg_str_resolve resolve 0) ^ " with
         Struct "^ s ^ " which has type " ^ (dbg_typ v1_type)))
      | [Ptr;Struct(s)], Ind -> MultiId_t(v1, Ind, v2, v2_type)
      | _ , _ ->
        raise (Failure ("Struct Error:
         Unknown struct with resolve operator " ^ (dbg_str_resolve resolve 0)
         ^ " and type" ^ (dbg_typ v1_type))))
| Binop (e1, op, e2) ->
  let lh = tc_expr e1 and rh = tc_expr e2 in
    let lh_type = get_type_lst_expr_t(lh)
    and rh_type = get_type_lst_expr_t(rh) in
    let ty = binop_result_type lh_type op rh_type in
      if lst_match ty [Err] then
       (* Binop_t(lh, op, rh, [Err]) *)
        raise (Failure ("Binop mismatch:
         Left side is " ^ (dbg_typ lh_type) ^ " Right
         side is " ^ (dbg_typ rh_type) ^
         " op is " ^ dbg_str_op op 0))
      else Binop_t(lh, op, rh, ty)
```

```
| Assign (s, e) ->
  let lh = (tc_expr s) and rh = (tc_expr e) in
    let lh_type = get_type_lst_expr_t(lh)
    and rh_type = get_type_lst_expr_t(rh) in
    let ty = assign_expr_result_type lh lh_type rh rh_type in
      if lst_match ty [Err] then
       (* Assign_t(lh, rh, [Err])*)
        raise (Failure ("Assign mismatch:
         Left side is " ^ (dbg_typ lh_type) ^ " Right
         side is " ^ (dbg_typ rh_type) ))
      else Assign_t(lh, rh, [Int])
| Call (fname, actuals) ->
  let param = List.map tc_expr actuals
  and rettyp = (get_func_entry fname).ret_ty in
  let decl_typs = get_func_decl_typs fname in
  let param_typs = get_typs_from_expr_t_lst param in
  if cmp_param_typ param_typs decl_typs fname then
    Call_t(fname, param, rettyp)
  else
    raise (Failure ("Function " ^ fname ^ " is using arguments of
    type " ^ (dbg_typ_ll param_typs) ^ " but its declaration uses type " ^
    (dbg_typ_ll decl_typs)))
| Pointer(e) -> let v1 = tc_expr e in
  let v1_type = get_type_lst_expr_t(v1) in
    Pointer_t(v1, (List.tl v1_type))
| Array(base,e) -> let v1 = tc_expr e in
  let b = tc_expr base in
  let v1_type = get_type_lst_expr_t(v1) in
    let btyp = get_type_lst_expr_t(b) in
    if is_int_or_char(v1_type) then
      Array_t(b, v1, (List.tl btyp))
    else
      raise (Failure ("Array index is type " ^ (dbg_typ v1_type) ^ " and not type int"))
     (*  Array_t(base, v1, [Err] @ btyp ) *)
| Addrof(e) -> let v1 = tc_expr e in
  let v1_type = get_type_lst_expr_t(v1) in
    Addrof_t(v1, [Ptr] @ v1_type)
| Negof(e) -> let v1 = tc_expr e in
  let v1_type = get_type_lst_expr_t(v1) in
    if is_int_or_char(v1_type) then
    Negof_t(v1, v1_type)
    else
      raise (Failure ("Wrong type " ^ (dbg_typ v1_type)
          ^ " for unary minus"))
    (* Negof_t(v1, [Err]) *)
| Noexpr -> Noexpr_t ([Void])
| Null -> Null_t ([Ptr;Void])
  in
```

```
let rec tc_stmt = function
    Block sl ->
    (List.fold_left (fun str lst -> str @ lst) [] (List.map tc_stmt sl) )
  | Expr e -> [Expr_t (tc_expr e)]
  | Return e ->
    let v1 = tc_expr e in
    let v1_type = get_type_lst_expr_t(v1) in
    let typ =  assign_result_type v1_type fdecl.ret in
    if typ = [Err] then
      raise (Failure ("Return type of function " ^ fdecl.fname ^
      " " ^ (dbg_typ fdecl.ret) ^ " does not match return type " ^
      (dbg_typ v1_type)))
    else
      [Return_t(tc_expr e)]
  | If (p, t, f) ->
    let v1 = tc_expr p and v2 = tc_stmt t and v3 = tc_stmt f in
    let v1_type = get_type_lst_expr_t(v1) in
      if is_int_or_char(v1_type) then
          [If_t(v1, Block_t(v2), Block_t(v3))]
      else
        raise (Failure ("If condition is type "
      ^ (dbg_typ v1_type) ^ " and not type int"))
  | While (e, b) ->
    let v1 = tc_expr e and v2 = tc_stmt b  in
    let v1_type = get_type_lst_expr_t(v1) in
      if is_int_or_char(v1_type) then
         [While_t(v1, Block_t(v2))]
      else
        raise (Failure ("While condition is type "
      ^ (dbg_typ v1_type) ^ " and not type int"))
  | For (asn, cmp, inc, b) ->
    let asn_t = tc_expr asn and cmp_t = tc_expr cmp
    and inc_t = tc_expr inc and stm_t = tc_stmt b in
    [For_t(asn_t, cmp_t, inc_t, Block_t(stm_t))]
in

let stmtblock = (tc_stmt (Block fdecl.body)) in

let rec has_return stmt_lst =
  match stmt_lst with
  | Return_t( _ )::tl -> true
  | _ ::tl -> has_return tl
  | [] -> false
in

(* Check return stmt exists if return type was declared  *)
(*if not(fdecl.ret = [Void]) && not(has_return stmtblock) then
  raise (Failure ("Function " ^ fdecl.fname ^ ", has return type " ^
```

```
  (dbg_typ fdecl.ret) ^ " but no return statement found"))
else *)
  [Sast(fdecl.fname, (conv2_expr_t fdecl.formals), stmtblock) ]
in

let env = { function_index = function_indexes;
                           global_index   = global_indexes;
               struct_index   = struct_indexes;
                           local_index    = StringMap.empty
             }
in

(* Code executed to start the program *)
let entry_function = try
  (StringMap.find "main" function_indexes); []
  with Not_found -> raise (Failure ("no \"main\" function"))
in

(* Compile the functions *)
(prog, List.concat (entry_function :: List.map (type_check_func env) functions));;
```

8.11 MAKEFILE

```
OBJS = ast.cmo parser.cmo scanner.cmo debug.cmo bytecode.cmo typecheck.cmo compile.cmo
execute.cmo cpi.cmo


TARFILES = Makefile testall.sh scanner.mll parser.mly \
        ast.ml sast.ml bytecode.ml debug.ml typecheck.ml compile.ml execute.ml cpi.ml \
        $(TESTS:%=tests/test-%.mc) \
        $(TESTS:%=tests/test-%.out)

cpi : $(OBJS)
        ocamlc -g -o cpi $(OBJS)

all : clean cpi

.PHONY : test
test : cpi
        cd tests && ./runtests.sh

test_tc : cpi
        cd tc_tests && ./runtests.sh

test_edpi :
        ssh edpi 'cd plt2013; make clean; git pull; make test'

test_tc_edpi :
```

```
	ssh edpi 'cd plt2013; make clean; git pull; make test_tc'

test_qemupi :
	ssh qemupi 'cd plt2013; make clean; git pull; make test'

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -g -c $<

%.cmi : %.mli
	ocamlc -g -c $<

microc.tar.gz : $(TARFILES)
	cd .. && tar czf microc/microc.tar.gz $(TARFILES:%=microc/%)

.PHONY : clean
clean :
	rm -f cpi parser.ml parser.mli scanner.ml testall.log \
	*.cmo *.cmi *.out *.diff
	rm -rf tests/out
	rm -rf tc_tests/*.s

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
sast.cmo: ast.cmo
sast.cmx: ast.cmx
bytecode.cmo: ast.cmo
bytecode.cmx: ast.cmx
debug.cmo: bytecode.cmo sast.cmo
debug.cmx: bytecode.cmx sast.cmx
typecheck.cmo: sast.cmo ast.cmo
typecheck.cmx: sast.cmx ast.cmx
compile.cmo: bytecode.cmo sast.cmo ast.cmo
compile.cmx: bytecode.cmx sast.cmx ast.cmx
execute.cmo: bytecode.cmo ast.cmo
execute.cmx: bytecode.cmx ast.cmx
cpu.cmo: scanner.cmo parser.cmi execute.cmo compile.cmo \
    bytecode.cmo ast.cmo type_check.cmo
cpi.cmx: scanner.cmx parser.cmx execute.cmx compile.cmx \
    bytecode.cmx ast.cmx typecheck.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
```

```
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
```