C_{\Pi}

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 ......................................................................................................................... 5

1.1  Key Features ......................................................................................................................... 5

1.2  Unsupported features from ANSI C: ...................................................................................... 6

2  Language Tutorial .................................................................................................................. 7

2.1  Programming in Cpi compared to ANSI C ............................................................................ 7

   2.1.1  Scoping Rules .................................................................................................................... 7

2.2  Common Substitutes to Implementing C features in Cpi .................................................... 7

3  Language Reference Manual .................................................................................................. 10

   Lexical conventions .................................................................................................................. 10

   Objects and lvalues .................................................................................................................. 11

   Expressions ................................................................................................................................ 11

      Identifier .................................................................................................................................. 11

      Constant ................................................................................................................................... 12

      String ....................................................................................................................................... 12

      ( expression ) .............................................................................................................................. 12

      primary-expression [ expression ] ............................................................................................. 12

      primary-expression ( expression-listopt ) .................................................................................. 12

      primary-lvalue . member-of-structure ......................................................................................... 13

      primary-expression -> member-of-structure .............................................................................. 13

   Unary operators .......................................................................................................................... 13

      * expression ............................................................................................................................... 13

      & lvalue-expression ................................................................................................................... 13

      - expression .............................................................................................................................. 13

   Multiplicative operators .......................................................................................................... 14

      expression * expression ............................................................................................................. 14

      expression / expression .......................................................................................................... 14

      expression % expression .......................................................................................................... 14

   Additive operators ..................................................................................................................... 14

      expression + expression .......................................................................................................... 14

      expression - expression .......................................................................................................... 14

   Relational operators ............................................................................................................... 15
4.4 Software Development Environment ................................................................. 28
4.5 Roles and responsibilities .................................................................................. 28
5 Architectural Design ............................................................................................. 29
  5.1 Overview .......................................................................................................... 29
  5.2 Scanning, Parsing, and AST .............................................................................. 30
  5.3 SAST Creation .................................................................................................. 32
  5.4 Bytecode Generation ....................................................................................... 33
  5.5 ARM Assembly Information ............................................................................ 35
6 Test Plan ................................................................................................................ 35
  6.1 Automation Script ............................................................................................ 35
  6.2 Test Case Selection .......................................................................................... 35
  6.3 Feature Tests Suite: .......................................................................................... 36
  6.4 Typechecking Test Suite .................................................................................. 36
  6.5 Demo Programs ............................................................................................... 37
7 Lessons Learned ................................................................................................... 63
  7.1 Edward Garcia ................................................................................................. 63
    Sean Yeh ......................................................................................................... 64
  7.2 Naveen Revanna .............................................................................................. 64
  7.3 ......................................................................................................................... 65
  7.4 Niket Kandya ................................................................................................. 65
8 Appendix ............................................................................................................... 65
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

Cπ will 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

<table>
<thead>
<tr>
<th>int</th>
<th>char</th>
<th>void</th>
<th>struct</th>
<th>NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>while</td>
<td>if</td>
<td>else</td>
<td>for</td>
<td>void</td>
</tr>
</tbody>
</table>

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
\cdot For accessing structure members through a structure variable
\rightarrow 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 or equal to operator
<= less than or equal to 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 C++.

```c++
/* **********************************************************
* 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
    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.

| switch( i ) | if (i== -1){             | if (i == -1){             |
|            |      n = n + 1; }         |      n = n + 1; }         |
| case -1:   | } else if (i == 0){       | } else if (i == 2){       |
|            |      z = z + 1; }         |      z = z + 1; }         |
| case 0 :   | } else if( i == 1){       | } else if (i == 3){       |
|            |      p = p + 1; }         |      p = p + 1; }         |
| case 1 :   | } else{                   | } else {                   |
|            |      i = i + 1; }         |      i = i + 1; }         |
| default:   | }                         | }                         |
| i = i + 1; | }                         | }                         |

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 \( E_1 \) be of pointer type, the expression \( \text{E}_1 -> \text{MOS} \) is exactly equivalent to \( (*\text{E}_1).\text{MOS} \).

**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.
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.
**TYPE SPECIFIERS**
The type-specifiers are

\[ \text{type-specifier:} \]

- 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.

\[ \text{declarator-list:} \]

- declarator
- declarator , declarator-list

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

\[ \text{declarator:} \]

- identifier
- * declarator
- declarator ( )
- declarator [ constant-expressionopt ]
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 \( \ldots \)”, where “\( \ldots \)” is the type which the identifier would have had if the declarator had been simply \( D \).

If a declarator has the form \( D[\quad] \) then the contained identifier has the type “function returning \( \ldots \)”, where “\( \ldots \)” is the type which the identifier would have had if the declarator had been simply \( D \).

A declarator may have the form

\[
D[\text{constant-expression}] \text{ 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 “\( \ldots \)”, then the declarator “\( D[\ i\ ] \)” yields a 1-dimensional array with rank \( i \) of objects of type “\( \ldots \)”.

**STRUCTURE DECLARATIONS**

Recall that one of the forms for a structure specifier is

\[
\text{struct } \{ \text{type-decl-list} \}
\]

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

\[
\text{type-decl-list:}
\]

\[
\text{type-declaration}
\]

\[
\text{type-declaration type-decl-list}
\]

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).
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 ;
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.

```c
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 \(\rightarrow\) 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.

```c
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

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

![Project Timeline](image)

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

**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 version 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

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

**TABLE 3. EXPRESSIONS USED IN AST CREATION**

<table>
<thead>
<tr>
<th>Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block</strong> (statement list)</td>
</tr>
<tr>
<td><strong>Expr</strong> (expression)</td>
</tr>
<tr>
<td><strong>Return</strong> (expression)</td>
</tr>
<tr>
<td><strong>If</strong> (expression, true_statement, false_statement)</td>
</tr>
<tr>
<td><strong>For</strong> (asn_expr, check_expr, incr_expr)</td>
</tr>
<tr>
<td><strong>While</strong> (condition_expr, statement_block)</td>
</tr>
</tbody>
</table>

**TABLE 4. STATEMENTS USED IN AST CREATION**
5.3 SAST Creation

Figure 4 shows 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.

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

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**

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 function 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.

<table>
<thead>
<tr>
<th>Atomics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lit (Literal_value)</td>
</tr>
<tr>
<td>Cchar (char)</td>
</tr>
<tr>
<td>Sstr(string_value,label)</td>
</tr>
<tr>
<td>Lvar(offset,size)</td>
</tr>
<tr>
<td>Gvar(name,size)</td>
</tr>
<tr>
<td>Pntr(addr,size)</td>
</tr>
<tr>
<td>Addr(atom)</td>
</tr>
<tr>
<td>Neg(atom)</td>
</tr>
<tr>
<td>Debug(debug_string)</td>
</tr>
</tbody>
</table>

**TABLE 7. ATOMIC USED IN BYTECODE GENERATION**
TABLE 8. BYTECODE STATEMENTS

<table>
<thead>
<tr>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom (atom)</td>
</tr>
<tr>
<td>VarArr (atom, atom)</td>
</tr>
<tr>
<td>Rval (atom)</td>
</tr>
<tr>
<td>BinEval (atom, atom, operation, atom)</td>
</tr>
<tr>
<td>BinRes (cpitypes list)</td>
</tr>
<tr>
<td>Assgmt (left_hand, right)hand)</td>
</tr>
<tr>
<td>Fcall (func_name, atom list, atom)</td>
</tr>
<tr>
<td>Branch (label)</td>
</tr>
<tr>
<td>Predicate (var_to_check, jump_on_what?, label)</td>
</tr>
<tr>
<td>Label (label_name)</td>
</tr>
</tbody>
</table>

Let's 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 representation 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.

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

6.4 Typechecking Test Suite
charptr.pass functions.pass if.pass
## 6.5 Demo Programs

### 6.5.1 Makefile

**bf:**

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

**bf_test:**

```
../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
, command:%c
",index,command); */
    /* printf("cell[0]: %d, cell[1]: %d
",dh[0], dh[1]); */
    /* printf("dh_index:%d
",*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"          1r]
    .text
    .global main

.main:
        stmfd sp!, {fp,

        bl   scanf
        str  r0, [fp,#-

        b1   scanf
        str  r0, [fp,#-

        bl   scanf
        str  r0, [fp,#-

        ldr  r0, [fp,#-

        ldr  r0, [fp,#-

        sub  r0, fp,#1264

        sub  r0, fp,#512

        sub  r0, fp,#512

        .LC1:
    .asciz "%s"

        add  fp, sp,#4

        ldr  r0, [fp,#-

        ldr  r0, [fp,#-

        ldr  r0, [fp,#-

        ldr  r0, [fp,#-

        ldr  r0, [fp,#-

        sub  r0, fp,#628

        str  r0, [fp,#-

        sub  r0, fp,#640

        .LC2:
    .asciz "len:

        .LC3:
    .asciz "\n"

        ldr  r0, [fp,#-

        ldr  r0, [fp,#-

        ldr  r0, [fp,#-

        ldr  r0, [fp,#-

        ldr  r0, [fp,#-

        sub  r0, fp,#128

        sub  r0, fp,#128

        sub  r0, fp,#128
blprintf
str r0, [fp,-1256]
str r0, [fp,-1256]
ldr r0, [fp,-620]
str r0, [fp,-620]
ldr r0, [fp,-620]
lr
add fp, sp,#4
sub sp, sp,#184
ldr r0, [fp,#-28]
ldr r1, [fp,-29]
ldr r1, [fp,-29]
ldr r0, [fp,-1260]
ldr r0, [fp,-1260]
ldr r0, [fp,-1260]
lr
add fp, sp,#4
sub sp, sp,#184
ldr r0, [fp,#-28]
ldr r1, [fp,#-29]
ldr r2, [fp,#-36]
ldr r3, [fp,#-44]
ldr r4, [fp,#-40]
ldr r1, [r4,#0]
ldr r0, [r4,#0]
ldr r1, [r4,#0]
add fp, sp,#4
sub sp, sp,#184
ldr r0, [fp,#-28]
ldr r1, [fp,#-29]
ldr r2, [fp,#-36]
ldr r3, [fp,#-44]
ldr r4, [fp,#-40]
ldr r1, [r4,#0]
ldr r0, [r4,#0]
ldr r1, [r4,#0]
lr
ladbf r1, [fp,#-1260]
ladbf r0, [fp,#-1256]
ladbf r1, [fp,#-1260]
ladbf r0, [fp,#-1256]
ladbf r1, [fp,#-1260]
ladbf r0, [fp,#-1256]
ladbf r1, [fp,#-1260]
add fp, sp,#4
sub sp, sp,#184
ldr r0, [fp,#-28]
ldr r1, [fp,#-29]
ldr r2, [fp,#-36]
ldr r3, [fp,#-44]
ldr r4, [fp,#-40]
ldr r1, [r4,#0]
ldr r0, [r4,#0]
ldr r1, [r4,#0]
lr
ladbf r1, [fp,#-1260]
ladbf r0, [fp,#-1256]
str r0, [fp,-620]
str r0, [fp,-620]
str r0, [fp,-620]
lr
add fp, sp,#4
sub sp, sp,#184
ldr r0, [fp,#-28]
ldr r1, [fp,#-29]
ldr r2, [fp,#-36]
ldr r3, [fp,#-44]
ldr r4, [fp,#-40]
ldr r1, [r4,#0]
ldr r0, [r4,#0]
ldr r1, [r4,#0]

do_command:

stmfd sp!, {fp,
lr
add fp, sp,#4
sub sp, sp,#184
ldr r0, [fp,#-28]
ldr r1, [fp,#-29]
ldr r2, [fp,#-36]
ldr r3, [fp,#-44]
ldr r4, [fp,#-40]
ldr r1, [r4,#0]
ldr r0, [r4,#0]
ldr r1, [r4,#0]
b do_command_exit
end1:
  ldrb r0, [fp,#-29]
  ldrb r1, [fp,#-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,#-68]
  ldr r8, [r4,#0]
  ldr r1, =1
  subs r3, r0, r1
  str r3, [fp,#-68]
  ldr r8, [fp,#-68]
  ldr r4, [fp,#-40]
  str r0, [r4,#0]
  str r8, [r4,#0]
  b do_command_exit
end2:
  ldrb r0, [fp,#-69]
  ldrb r1, [fp,#-69]
  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,#-69]
  ldr r0, [fp,#-76]
  ldr r4, [fp,#-16]
  strb r0, [r4,#0]
  ldr r8, [fp,#-44]
  b do_command_exit
end3:
  ldrb r0, [fp,#-29]
  ldrb r1, [fp,#-45]
  cmp r0, r1
  moveq r3,#1
  movne r3,#0
  uxtb r3,r3
  strb r3, [fp,#-93]
  ldrb r0, [fp,#-93]
  cmp r0,#0
  beq end4
  ldr r4, [fp,#-16]
  ldr r1, [fp,#-16]
  b1 scanf
  str r0, [fp,#-]
  b do_command_exit
  end4:
  ldrb r0, [fp,#-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 end5
  ldr r4, [fp,#-44]
  ldr r0, [fp,#-12]
  ldr r4, [fp,#-16]
  ldr r0, [r4,#0]
  ldr r1, =0
  cmp r0, r1
  moveq r3,#1
  movne r3,#0
  uxtb r3,r3
  str r3, [fp,#-]
  b do_command_exit
end5:
  str r0, [fp,#-108]
  ldr r0, [fp,#-108]
cmp r0,#0          uxtb r3,r3          movne r3,#0
beq else12        strb r3, [fp,#-]     uxtb r3,r3
ldr r0, =0        ldrb r0, [fp,#-]     strb r3,r3
125]             141]             141]
str r0, [fp,#-24] ldrb r0, [fp,#-]     ldrb r0, [fp,#-]
ldr r0, =0        cmp r0,#0          cmp r0,#0
str r0, [fp,#-20] beq end7          beq end7
b loop2_end
loop2_start:

ldr r0, [fp,#-44] ldr r0, [fp,#-24] ldr r0, [fp,#-24]
ldr r1, =1        ldr r1, =1        ldr r1, =1
end7:             end8:             end10:
adds r3, r0, r1   adds r3, r0, r1   loop2_end:
str r3, [fp,#-]   ldr r0, [fp,#-]   ldr r0, [fp,#-]
112]             148]             148]
l dr r0, [fp,#-]   cmp r0, r1        movne r3,#0
str r0, [fp,#-44] movne r3,#0        str r0, [fp,#-24]
l dr r0, =1        uxtb r3,r3        end11:
ldr r1, [fp,#-44] str r3, [fp,#-]   end11:
l dr r1, =1        loop2_end:
muls r3, r0, r1   ldr r0, [fp,#-20] ldr r0, [fp,#-20]
132]             132]             132]
l dr r0, [fp,#-36] cmp r0,#0        cmp r0,#0
ldr r1, [fp,#-]   beq end11        moveq r3,#1
116]             152]             152]
adds r3, r0, r1   ldrb r0, [fp,#-5] movne r3,#0
str r3, [fp,#-]   ldrb r1, =93     uxtb r3,r3
120]             152]             152]
l dr r4, [fp,#-]   movne r3,#0        cmp r0,#0
ustrb r3,r3       cmp r0,#0        moveq r3,#1
120]             152]             152]
l rdb r0, [r4,#0] ldr r0, [fp,#-24] cmp r0,#0
strb r0, [fp,#-5] ldr r1, =1        moveq r3,#1
133]             112]
l dr r0, [fp,#-5] ldrb r0, [fp,#-]     bne loop2_start
ldr r1, =0        ldr r0, [fp,#-12] b end12
133]             124]     else12:
l dr r0, [fp,#-24] cmp r0,#0        loop4_start:
ldr r0, [fp,#-]   beq end9        ldr r0, [fp,#-12]
140]             124]        str r0, [fp,#-12]
ldr r0, [fp,#-]   ldr r0, [fp,#-12] ldr r0, [fp,#-44]
140]             124]        str r0, [fp,#-44]
ldr r0, [fp,#-]   str r3, [fp,#-]   str r3, [fp,#-]
156]             124]             124]
ldr r0, [fp,#-]   str r0, [fp,#-24] cmp r0,#1
156]             124]        str r0, [fp,#-24]
ldr r0, [fp,#-]   ldr r0, [fp,#-5]     end9:
ldr r0, =5        ldr r0, [fp,#-5]     end9:
ldr r1, =93      ldr r1, =93        end9:
l dr r1, =93      ldr r1, =93        end9:
l dr r1, =93      ldr r1, =93        end9:
l dr r0, r1       ldr r0, r1
moveq r3,#1      ldr r0, r1
movne r3,#0       movne r3,#1
muls r3, r0, r1   muls r3, r0, r1
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;
        *last = *head;
    }
}

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\n");
    printf("1.) 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);
        } else if (i == 3){
            display(&head, &last);
        } else if (i == 4){
            break;
        } else {
            printf("Invalid choice. Please try again.\n");
        }
    }
}

void insert_beginning(int value, struct node **head, struct node **last)
{
    struct node *temp;
    temp = (struct node *)malloc(sizeof(struct node));
    if (temp == NULL)
    {
        printf("Allocation failed\n");
        return;
    }
    temp->data = value;
    temp->next = *head;
    *head = temp;
    if (*last == NULL)
    {
        *last = temp;
    }
}

void delete_from_end(struct node **head, struct node **last)
{
    if (*last == NULL)
    {
        printf("List is Empty!\n");
        return;
    }
    struct node *temp, *prev;
    temp = *last;
    if (*last->next == NULL)
    {
        *head = *last = NULL;
        return;
    }
    prev = *head;                
    while (temp->next != NULL)
    {
        prev = temp;
        temp = temp->next;
    }
    prev->next = NULL;
    free(temp);
    if (*last == temp)
    {
        *last = prev;
    }
}

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;
    }
}
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\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
.globl 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]
bl loop1_start:  
    ldr r0, .LC5
    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 printf
    str r0, [fp,#-64]
    ldr r0, [fp,#-8]
    sub r1, fp,#20
    sub r2, fp,#24
    b insert_beginning

else4:

loop1_start:  
    ldr r0, .LC5
    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 printf
    str r0, [fp,#-64]
    ldr r0, [fp,#-8]
    sub r1,fp,#20
    sub r2, fp,#24
    b display

else4:

loop1_start:  
    ldr r0, .LC5
    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 printf
    str r0, [fp,#-64]
    ldr r0, [fp,#-8]
    sub r1,fp,#20
    sub r2, fp,#24
    b display

else4:
```assembly
ldr r0, [fp, #-12]  #0  str r3, [fp, #20]  ldr r0, [fp, #-20]
ldr r1, #2  b main_exit  ldr r0, [fp, #-20]
cmp r0, r1  b end1  cmp r0,#0
moveq r3,#1  else1:  beq end5
movne r3,#0  ldr r0, #0  ldr r0, .LC9
uxtb r3,r3  b main_exit
str r3, [fp, #68]  end1:  bl printf
end2:  str r0, [fp, #-28]
}
cmp r0,#0  end3:  loop2_start:
beq else3  end4:
sub r0, fp,#20  loop2_end:
sub r1, fp,#24  loop1_start:
bl delete_from_end  main_exit:
\begin{verbatim}
delete_from_end
str r0, [fp, #72]  loop1_start:
sub r0, fp,#20  sub sp, fp, #4
sub r1, fp,#24  ldmfd sp, [fp, #44]
\end{verbatim}
bl display  .data
b end3  .LC9:
\begin{verbatim}
display:
\end{verbatim}
else3:
\begin{verbatim}
ldr r0, [fp, #12]  Empty!
ldr r1, #3  .LC10:
str r0, r1
add s3, r0, r1
moveq r3,#1
movne r3,#0
uxtb r3,r3
str r3, [fp, #76]
\end{verbatim}
\begin{verbatim}
"List is"
"-> %d"
.display:
stmfd sp, {fp, lr}
add fp, sp,#4
sub sp, fp, #44
str r0, [fp,#12]
str r1, [fp,#16]
ldr r4, [fp,#12]  loop1_start:
ldr r0, [r4,#0]
moveq r3,#0
movne r3,#1
uxtb r3,r3
\end{verbatim}
else2:
\begin{verbatim}
ldr r0, [fp, #12]  loop2_end:
ldr r0, [fp,-8]
ldr r1, #0
add s3, r0, r1
moveq r3,#0
movne r3,#1
uxtb r3,r3
\end{verbatim}
\begin{verbatim}
ldr r1, #4
ldr r0, #0
cmp r0, r1
muls r3, r0, r1
moveq r3,#0
movne r3,#1
uxtb r3,r3
\end{verbatim}
\begin{verbatim}
ldr r0, #12
ldr r1, #0
mul s3, r0, r1
moveq r3,#1
movne r3,#0
\end{verbatim}
```

.data
.LC11:
    .asciz "Cannot delete from list is %d 
"
.LC12:
    .asciz "\nData deleted from list is %d 
"
.LC13:
    .asciz "\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, #12]
    ldr r4, [fp, #12]
    ldr r0, [r4, #0]
    str r0, [fp, #12]
    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, #8]
    ldr r0, [fp, #8]
    cmp r0, #0
    beq end6
    str r0, [fp, #8]
    ldr r0, [fp, #8]
    str r0, [fp, #8]
    cmp r0, #0
    beq end6
    str r0, [fp, #8]
    ldr r0, #1
    ldr r1, [r4, #0]
    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
    str r0, [fp, #20]
    ldr r0, #0
    ldr r4, [fp, #12]
    str r0, [r4, #0]
    .text
    .asciz "Adding to Empty List\n"
.LC14:
    ldr r0, #0
    str r0, [r4, #0]
    .asciz "Adding to List\n"
.LC15:
    ldr r4, [fp, #16]
    str r0, [r4, #0]
    ldr r0, #0
    .text
    .global insert_beginning
    insert_beginning:
printf("|%c|%c|%c|
", board[0], board[1], board[2]);

int printboard(char board[]){
    printf("-------\n");
    printf("|%c|%c|%c|
", board[6], board[7], board[8]);
}

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

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\n");
    printf("Valid inputs are 0-9n\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
wins!
"Player %d!
"inputs are 0-9\n\n1: 'O'
Player 2: 'X'

lr
.global
.text

main:
  stmfd sp!
  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
  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
  strb r0, [r4,#0]
  ldr r1, =8
  muls r3, r0, r1
  str r3, [fp,#-64]
  ldr r1, =5
  muls r3, r0, r1
  str r3, [fp,#-64]
  adds r3, r0, r1
  str r3, [fp,#-76]
  ldrb r0, #32
  ldr r1, [fp,#-76]
  ldr r0, =1
  strb r0, [r4,#0]
  ldr r1, =8
  muls r3, r0, r1
  str r3, [fp,#-64]
  ldr r1, =5
  muls r3, r0, r1
  str r3, [fp,#-64]
  adds r3, r0, r1
  str r3, [fp,#-76]
  ldrb r0, #32
str r3, [fp,#-]
lldr r0, #32
ldr r4, [fp,#-

strb r0, [r4,#0]
ldr r0, #1
ldr r1, #9
muls r3, r0, r1
str r3, [fp,#-

sub r0, fp,#40
ldr r1, [fp,#-

adds r3, r0, r1
str r3, [fp,#-

ldrb r0, #32
ldr r4, [fp,#-

strb r0, [r4,#0]
ldr r0, [fp,#-

ldrb r0, #32
ldr r4, [fp,#-

sub r0, fp,#40
ldr r1, [fp,#-

cmp r0, #0
adds r3, r0, r1
movl r3,#1
ldr r4, [fp,#-

ldr r0, [fp,#-
str r0, [fp,#-

ldr r0, #0
str r0, [fp,#-

ldr r0, #0
str r0, [fp,#-

ldr r0, #0
str r0, [fp,#-

ldr r0, [fp,#-

str r0, [fp,#-

end1:

loop2_start:
sub r0, fp,#40
str r0, [fp,#-

bl printboard
str r0, [fp,#-

loop1_start:

loop2_end:

loop1_end:
ldr r3, [fp, #-12] 294
movlt r0, r3

ldr r0, [fp, #0] 292
cmp r0, #0
beq end5

sub r0, fp, #40
1dr r0, [fp, #8]
bd getchar

bl r0, [fp, #12] 308

strb r0, [fp, #0] 288
ldr r0, [fp, #0]
movne r3, #1

add2 r3, r0, r1
str r3, [fp, #0]

ldrb r0, [fp, #0] 300
ldr r0, [fp, #0]

strb r0, [fp, #0] 308
ldr r0, [fp, #28]
add2 r3, r0, r1
str r3, [fp, #0]

ldrb r0, [fp, #0] 296
ldr r1, [fp, #8]
cmp r0, r1
moveq r3, #1

ldr r0, [fp, #0] 304

add2 r3, r0, r1
str r3, [fp, #0]

ldrb r0, [fp, #0] 300
ldr r0, [fp, #8]

str r0, [fp, #12] 316
ldr r0, [fp, #12]

ldrb r0, [fp, #0] 320
ldr r0, [fp, #12]

str r0, [fp, #12]
ldr r0, [fp, #12]

ldrb r0, [fp, #0] 324
ldr r1, [fp, #28]
add2 r3, r0, r1
str r3, [fp, #0]

ldr r1, [fp, #0] 308
ldr r0, [fp, #28]
cmp r0, r1
bne loop2_start

movge r3, #1

movle r3, #0

movlt r3, #0

loop2_end:
end6:

else6:

sub r0, fp, #40
bl checkboard

ends:

main_exit:

sp

lbf sp

fp

pc

plwz r3, #0

ldr r0, [fp, #0]

cmp r0, #0

beq end5

sub r0, fp, #40

1dr r0, [fp, #40]

lbf r3, [fp, #12] 300

str r3, [fp, #12] 316

ldr r0, [fp, #12]

str r3, [fp, #12]

ldr r0, [fp, #12]

ldr r0, [fp, #12]

loop2_start:

main_exit:

sp

lbf sp

fp

pc
.data

.mtext

.global getchar

getchar:
    stmfd sp!, {fp, lr}
add fp, sp, #4
sub sp, sp, #8
str r0, [fp, #-8]
ldr r0, [fp, #-8] 53
ldr r1, [fp, #-12] 53
cmp r0, r1
moveq r3, #1
movne r3, #0
uxtb r3, r3
str r0, [fp, #-8] 53
ldr r0, [fp, #-12]
add r3, r0, r1
add r3, r0, r1
beq end9
ldr r0, [fp, #179]
bl checkcol_exit
end9:
    ldrb r0, #198
b getchar_exit
getchar_exit:
    sub sp, fp, #4
ldmfd sp!, {fp, pc}
    loop3_end:
    ldr r0, [fp, #-12]
lhr r0, [fp, #9]
cmp r0, r1
movlt r3, #1
movge r3, #0
movne r3, #0
uxtb r3, r3
str r3, [fp, #37944]
ldr r0, [fp, #37944]
add r3, r0, r1
add r3, r0, r1
ldr r0, [fp, #8]
ldr r1, [fp, #8]
ldr r0, [fp, #8]
loop3_start:
    ldr r0, #3
ldr r1, [fp, #-12]
adds r3, r0, r1
str r3, [fp, #-84]
ldr r0, =1
ldr r1, =8
mul 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, [fp, #-16]
ldr r1, [fp, #-16]

muls r3, r0, r1

cmp r0, #0
beq end15
adds r3, r0, r1
str r3, [fp, #-16]

ldrb r0, [fp, #-28]
l1dr r1, [fp, #-16]
adds r3, r0, r1
str r3, [fp, #-28]

ldrb r0, [r4, #0]

ldrb r1, [r4, #0]

cmp r0, r1
moveq r3, #1
movne r3, #0
u3xtb r3, r3
strb r3, [fp, #-92]

ldrb r4, [fp, #-92]

ldrb r0, [fp, #0]

cmp r0, #0
beq end10
ldr r0, =1

str r0, [fp, #-8]

drd10:

ldrb r0, [fp, #16]
ldr r1, [fp, #16]
adds r3, r0, r1
str r3, [fp, #16]

end11:

end12:

ldr r0, =1

ldr r1, =2
mul r3, r0, r1
str r3, [fp, #-16]

ldr r0, [fp, #16]

ldr r1, [fp, #16]

adds r3, r0, r1
str r3, [fp, #16]

100:

ldr r0, [fp, #28]

ldr r1, [fp, #28]

adds r3, r0, r1
str r3, [fp, #28]

104:

ldr r4, [fp, #28]

ldr r0, [fp, #28]

checkboard_exit:

sub sp, fp, #4
ldmfd sp!, {fp, pc}

mul r3, r0, r1

.data
.LC7:

.asciz "Column win!"

.text

.global checkcol
checkcol:
  stmfd sp!, {fp, lr}
  add fp, sp, #4
  sub sp, fp, #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] 65
  ldr r0, #1
  ldr r1, [fp, #-8] 65
  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, #1
  movne r3, #1
  moveq r3, #0
  beq end17
  ldr r0, [fp, #-16]
  str r0, [fp, #0]
  muls r3, r0, r1
  str r3, [fp, #-20]
  ldr r0, [fp, #-20]
  ldr r1, #1
  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
  movne r3, #0
  moveq r3, #0
  ldrb r0, [fp, #-20]
  ldrb 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] 65
ldr r0, =1
ldr r1, [fp, #36] 65
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]
ldr r1, [r4, #0]
cmp r0, r1
moveq r3, #1
movne r3, #0
uxtb r3, r3
strb r3, [fp, # -84] 45
ldrb r0, [r4, #0] 45
ldrb r0, [fp, # -84] 45
cmp r0, #0
beq end21
ldr r0, =1
ldr r1, [fp, #8] 77
muls r3, r0, r1
str r3, [fp, #52]
ldr r0, [fp, #16]
ldr r1, [fp, #52]
bl printf
adds r3, r0, r1
str r3, [fp, #56]
ldr r0, =1
ldr r1, [fp, #12] 77
cmp r0, #0
beq end19
ldr r0, =1
.m.C8
adds r3, r0, r1
str r3, [fp, #56]
ldr r0, =1
b checkrow_exit
ldr r1, [fp, #12]
end19:
muls r3, r0, r1
str r3, [fp, #60]
ldr r0, [fp, #16]
ldr r1, [fp, #60]
checkrow_exit:
sub sp, fp, #4
ldmfdr sp, {fp, pc}
ladr r4, [fp, #64]
ldrb r0, [r4, r0]
ldrb r1, [r4, #0]
.data
cmpeq r3, #1
movne r3, #0
uxtb r3, r3
strb r3, [fp, # -84]

.asciz
"|%c|%c|%c| \n"

.asciz
"-------

.asciz
"-------

.asciz
"-------

.asciz
"-------

.global printfboard
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
```c
#include <stdio.h>

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

    while(c < n) {
```

6.5.9 VARIABLEARRAY

```c
#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
  .asciz "b[%d] = %c\n"
  .global fun
fun:
  .text
  .global main
main:
  .data
  .global main
main:
  .data
  .global fun
fun:
  .text
  .global main
main:
  .data
  .global fun
fun:
  .text
  .global main
main:
  .data
  .global fun
fun:
  .text
  .global main
main:
  .data
  .global fun
fun:
  .text
  .global main
main:
  .data
  .global fun
fun:
  .text
  .global main
main:
ldr r0, #4
ldr r1, [fp, #-8]
ladr r0, [fp, #-52]
ldr r0, [fp, #-76]
ladr r0, #4
muls r3, r0, r1
cmp r0,#0
str r3, [fp, #-8]
bne loop1_start
ldr r0, [fp, #8]
ldr r0, r0, r1
add r3, r0, r1
str r3, [fp, #28]
str r0, [fp, #20]
adds r3, r0, r1
ldr r0, [fp, #8]
ldr r0, #0
ldr r1, [fp, #12]
str r0, [fp, #8]
ldr r0, #0
ldr r1, [fp, #12]
adds r3, r0, r1
str r3, [fp, #32]
loop2_start:
ldr r0, [r4, #0]
ldr r1, [fp, #8]
adds r3, r0, r1
str r3, [fp, #97]
ldr r1, [fp, #8]
muls r3, r0, r1
str r3, [fp, #56]
ldr r0, [fp, #12]
str r3, [fp, #88]
ldr r0, [fp, #88]
ldr r0, [fp, #88]
ldr r1, [fp, #56]
str r0, [fp, #20]
ldr r0, [fp, #20]
ldr r0, [fp, #8]
ldr r0, [fp, #8]
ldr r0, [fp, #8]
loop2_end:
loop1_start:
ldr r0, #4
ldr r1, [fp, #8]
adds r3, r0, r1
str r3, [fp, #4]
ldr r1, [fp, #4]
adds r3, r0, r1
muls r3, r0, r1
str r3, [fp, #56]
ldr r0, [fp, #12]
str r3, [fp, #88]
ldr r0, [fp, #88]
ldr r0, [fp, #88]
adds r3, r0, r1
str r3, [fp, #60]
ldr r1, [r4, #0]
str r0, [fp, #8]
ldr r1, [r4, #0]
ldr r4, [fp, #40]
ldr r1, [fp, #60]
str r0, [fp, #8]
ldr r1, [fp, #40]
add r3, r0, r1
str r3, [fp, #44]
ldr r0, [fp, #36]
ldr r4, [fp, #44]
ldr r0, [r4, #0]
ldr r0, [fp, #36]
ldr r1, [fp, #8]
muls r3, r0, r1
str r3, [fp, #56]
ldr r0, [fp, #12]
str r3, [fp, #88]
ldr r0, [fp, #88]
ldr r4, [fp, #40]
ldr r1, [fp, #60]
str r0, [fp, #8]
loop_end:
loop1_end:
ldr r0, [fp, #8]
ldr r1, [fp, #24]
cmp r0, #1
movlt r3,#1
mover r3,#0
str r3, [fp, #48]
ldr r0, [fp, #8]
ldr r1, [fp, #48]
add 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, [fp, #8]
muls r3, r0, r1
str r3, [fp, #68]
ldr r0, [fp, #16]
str r3, [fp, #96]
ldr r0, [fp, #96]
cmp r0,#0
str r3, [fp, #72]
bne loop2_start
ldr r0, [LC1]
ldr r1, [fp, #20]
bl fun_exit
fun_exit:
sub sp, fp, #4
ldmfd sp!, {fp, pc}
bl printf

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**

```plaintext
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
   | Add of expr
   | Neg 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 cpiotypes = Void | Int | Char | Ptr | Arr of expr | Struct of string | Err

type var_decl = {
    vname: string;
    vtype: cpiotypes 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 : cpiotypes 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) *)
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
let build_global_idx map = StringMap.empty;;
let gl_atm a = get_atom(List.hd (List.rev a));;

(* Calculates 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
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 =@; 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

(* 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"

let gen_binres_type typ =
  [BinRes(typ)]

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)
   )
  )

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"

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)

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"
  | _ -> ""

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
  | _ -> ""
let gen_atom atm = [Atom (atm)]

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

let incr_by_ptrsz exp incrsz tmp = [BinEval (tmp, (Lit incrsz), Mult, (gl_atm exp))]

let get_struct_table stct = (try (StringMap.find stct env.struct_index).memb_index with Not_found -> err " struct " ^ stct ^ " is not a type")

let gen_addr_lst v1 = v1 @ gen_atom (Addr(gl_atm v1))

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)))

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 @ id))
       (get_ptrsize_type hd.vtype) tmp) @
       [VarArr((get_lvar_varname env.local_index @ hd.vname), tmp)]
       | _ -> [])
| _ -> []) @ (gen_vararr tl)

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
  | _  -> (match list.hd v2binres with
  | _  -> BinEval (v3, (gl_atm v1), op, gl_atm v2))
  | _  -> err "Cannot reach here"
)
)

(*) 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)

let rec expr ?(table = env.local_index) ?(strict=0) = function
  | Literal i -> (gen_binres_type [Int]) @ gen_atom (Lit i)
  | String s ->
    let l1 = incr count_mem; ".LC" ^
    (string_of_int !count_mem) in
    (gen_binres_type [Ptr;Char]) @
    gen_atom(Sstr(s, l1))
  | 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
(mmatch 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 (g_l_atm v1),
(gen_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 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 (g_l_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 (g_l_atm v1)]
| If (p, t, f) ->
let pval = expr p and tval = stmt t and fval = stmt f in
let l0 = (g_l_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 (l1)]
@ [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 l3 = (g_l_atm v2) in
[Branch l1] @ [Label l0] @ v1 @ [Label l1] @ v2 @ [Predicate (v3, true, l0)]
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 CπML

type action = Ast | Interpret | Bytecode | Compile

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

(* 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, "Read from stdin"),
  ("--stdout", Arg.Set use_stdout, "Output to stdout"),
  ("-b", Arg.Set debug_bytecode, "Print out bytecode"),
  ("-sast", Arg.Set debug_sast, "Print out sast"),
  ("--binary", Arg.Set create_binary, "Create binary executable (only if -o is set)"),
  ("-o", Arg.String (fun x -> out_file := x), "Set output file"),
  ("-tc", Arg.Set no_sast, "Turn 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) -> "Negative: \n" ^ p (tabs+2) ^ "Value \n" ^ (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"
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 ^ "\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))
  | Assign (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
        [] -> ""
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 ^ "\n"
  ^ 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"
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:" ^ "\n"
   ^ 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_strResolve 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 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"))

let idx_to_offset off = off + size_stmfd

let rec print_atom_lst = function
  | [] -> ""
  | hd :: tl ->
    dbg_print hd ^ (print_atom_lst tl)

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 "")
                  ^ "^ " ^ sym ^ string_of_int (idx_to_offset off) "\n")
    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,#" ^ sym ^ string_of_int (idx_to_offset off) "]")
      | Gvar (vname, sz) -> "" (*TODO *)
      | Neg (vnm) -> p ("rsb " ^reg^ ", ^ reg ^", ^ #")
      | 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: " 1)
      | 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: " 1)
      | Sstr (s, l) -> p ( "ldr r0, =" ^ l)
      | Debug (s) -> s
    in
  gen_ldr_str_code "ldr" "=" reg var

and store_code reg var =
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"
    | Mul -> 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,#1" ^
      p "movne r3,#0" ^
      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
    | [] -> ""
 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") ^
    (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
" (List.fold_left
  (fun str lst -> str ^ (mem_code_gen lst))
  "" body) ^ "\n" ^
".text
" (* 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 %}

%{ 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 INDICTION 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 }

actuels_opt:
   /* nothing */ { [] }
   | actuels_list { List.rev $1 }

actuels_list:
   expr            { [$1] }
   | actuels_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 of expr_t * stmt_t * stmt_t
| For of expr_t * expr_t * expr_t * stmt_t
| While 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
    [' ' '	' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"    { comment lexbuf }           (* Comments *)
| "//"    { line_comment lexbuf }
| "#include" { includes lexbuf }
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
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]
let f3_index = 
StringMap.add "scanf"
{
  param = [];
  ret_ty = [Int]
} 
f2_index

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

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

(* 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))
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
    | _ _ -> 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::t1, [Int], Add, _ _ -> ty1
  | Ptr::t1, [Char], Add, _ _ -> ty1
  | Ptr::t1, [Int], Sub, _ _ -> ty1
  | Ptr::t1, [Char], Sub, _ _ -> ty1
  | [Int], Ptr::t1, Add, _ _ -> ty2
  | [Char], Ptr::t1, Add, _ _ -> ty2
  | [Int], Ptr::t1, Sub, _ _ -> ty2
  | [Char], Ptr::t1, Sub, _ _ -> ty2
  | Arr(s)::t1, [Int], Add, _ _ -> ty1
  | Arr(s)::t1, [Char], Add, _ _ -> ty1
  | Arr(s)::t1, [Int], Sub, _ _ -> ty1
  | Arr(s)::t1, [Char], Sub, _ _ -> ty1
  | [Int], Arr(s)::t1, Add, _ _ -> ty2
  | [Char], Arr(s)::t1, Add, _ _ -> ty2
  | [Int], Arr(s)::t1, Sub, _ _ -> ty2
  | [Char], Arr(s)::t1, Sub, _ _ -> ty2
  (* | Ptr::t1, [Int], Equal, _ _ -> ty1
  | Ptr::t1, [Char], Equal, _ _ -> ty1 *)
  | Pag::t1, [Ptr;Void], Equal, _ _ -> [Int]
  | Pag::t1, [Ptr;Void], Neq, _ _ -> [Int]
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 |
|     | [Ptr;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 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 " ^
  *)
let env = { function_index = function_indexes;
   global_index = global_indexes;
   struct_index = struct_indexes;
   local_index = StringMap.empty
}

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

(* 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