Polytechnic University of Bari

Department of electrical and information engineering

Master's degree course on

Computer Engineering – Information Systems

Project on

Formal Languages and Compilers

From C to WebAssembly

Professor:

Prof. Ing. Floriano Scioscia

Students:

G. Allegretta

F. Scarangella

Academic Year 2017/2018

Summary

Introduction

Aim of this project is to develop a front-end compiler which transform C source files into .wat files, representing an intermediate text format of WebAssembly as target language.
C files will be a subset of C99 standard (**ISO/IEC 9899:1999**).

C is an imperative procedural language. It was designed to be compiled using a relatively straightforward compiler, to provide low-level access to memory and language constructs that map efficiently to machine instructions and to require minimal run-time support. Despite its low-level capabilities, the language was designed to encourage cross-platform programming.
In particular, C99 represents a past version of the C programming language standard.
After ANSI produced the official standard for the C programming language in 1989, which became an international standard in 1990, the C language specification remained relatively static for some time.
Normative Amendment 1 created a new standard for C in 1995, but only to correct some details of the 1989 standard and to add more extensive support for international character sets.
The standard underwent further revision in the late 1990s, leading to the publication of ISO/IEC 9899:1999 in 1999, which was adopted as an ANSI standard in May 2000. The language defined by that version of the standard is commonly referred to as "C99".

WebAssembly (Wasm, WA) is a web standard that defines a binary format and a corresponding assembly-like text format for executable code in Web pages. It is meant to enable executing code nearly as fast as running native machine code. It was envisioned to complement JavaScript to speed up performance-critical parts of web applications and later on to enable web development in languages other than JavaScript. WebAssembly does not attempt to replace JavaScript, but to complement it. It is developed at the World Wide Web Consortium (W3C) with engineers from Mozilla, Microsoft, Google and Apple.

Restrictions

Within the project it has been chosen a subset of C99 grammar, which includes the following features:

Source language: C

Intermediate Representation: WAT (using S-Expression)

Supported constructs

- Sequence
- A branching construct (*if*)
- A Loop construct (*for*)
- An input construct (*scanf*)
- An output construct (*printf*)

Types allowed

- Integer values
- Float values
- Character values
- Array
- Struct
    - Typedef keyword to name structures is not supported
    - Accessing members of a structure is possible by using dotted notation
    - Accessing members by using structure pointer operator "->" is not supported

In addition, there will be considered functions with parameters passed by value.

Operators

- Arithmetic
    - Binary
      All four usual mathematical operators "+" "-" "*" "/"
    - Unary
      Reverse operation -a and postfix increment and decrement operators "++" "--"
- Relational (Are accepted only comparison between numbers)
    - Equal "=="
    - Not equal "!="

    - Less then “<”
    - Greater then “>”
    - Less or equal then “≤”
    - Greater or equal then “≥”
- Logical
    - And “&&”
    - Or “||”
    - Not “!”

Global C program

C is a procedural language, so the main structure of a generic source file is composed, in order, of inclusions of libraries, declarations of global variables and functions definition.
Single function definition is composed, in order, of variable declarations and statements, where statements include one or more assignment, function calls or generic instructions.

In our project it has been decided to not include prototype but only function definition and it’s mandatory to define main function as last function[1].
In addition, it does not accept statements outside functions scope..

[1] An example of not allowed syntax can be found at test/not_working/improper_program.c
An example of not allowed declaration can be found at test/not_working/struct.c

Lexical Analyzer

Within token definition section, regular expressions have been used to define whitespaces, numbers and alphabetics.
There are also defined two states for comment (both on single line and on multi line).

```
/* TOKEN DEFINITION BY USING REGULAR EXPRESSION */
/* comment state definition */
%x comm
%x line_comm
/* whitespaces */
delim           [ \t]
ws              {delim}+
/* numbers */
digit           [0-9]
exp             ([Ee][+-]?{digit}*)
/* alphabetics */
alpha           [a-zA-Z]
underscore      [_]
char            \'.\'
string          \"(\\.|[^"\\])*\"
header          #include{ws}<{alpha}*(\.{alpha}*)?>
id              {alpha}({alpha}|{digit}|{underscore})*
```

Comments and headers are recognized and discarded.

```
/* LIST OF TOKENS AND ACTIONS */
%%
"/*"                        BEGIN(comm);
<comm>[^*/\n]*              /* if not a '*' or new line, discard everything */
<comm>"*"+[^*/\n]*          /* discard '*' not warning for comment end and all next
characters */
<comm>\n                    /* discard new lines */
<comm>"*"+"/"               BEGIN(INITIAL);

"//"                        BEGIN(line_comm);
<line_comm>[^/\n]*          /* if not a new line, discard everything */
<line_comm>\n               {BEGIN(INITIAL);}
```

Type of informational content depends on the token itself and, generally, can be a char or an int value.

```
    static inline void value_token() { yylval.sval = strdup(yytext); }
```

This allows us to copy the value of the token, contained in the variable yytext.

At the beginning of the lex.l itself there is a line

```
%option noyywrap
```

This option is necessary because the scanner calls, by default, the function yywrap on the end of file.

C keywords have been defined before all other alphanumeric definitions.

Syntax Analyzer

In the first section of the Bison source code are included all the needed libraries containing macros and inclusions used, definition of functions used in the translation and definition of variables.

```
%{
    /* CONTENT TO BE COPIED AT THE BEGINNING */

    /* include directives */
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include <libgen.h>
    #include "../src/utils.h"
    #include "../src/ast.h"
    #include "../src/list.h"
    #include "../src/symtab.h"
    #include "../src/wat.h"

    int yylex();
    void yyerror (const char *s);
    /* Variable needed for debugging */
//  int yydebug = 1;

    extern FILE *yyin;
    AST *ast;                       // Abstract Syntax Tree
    SymTab *symtab;                 // Symbol Table
%}
%error-verbose
```

Then there are the declarations of yylex, yyerror and yydebug (useful for solving problems during parsing) followed by the declaration of 2 fundamental structures of the project: AST and Symbol Table.
At the end of Prolog section, supported rule types are defined by %union keyword, tokens and non-terminals are listed with the corresponding type and precedence rules are defined.

In the second section there are rules to detect whether the sequence of tokens respects or not a production. At each rule is associated an action written in C. This allows the construction of AST and Symbol Tables, used to generate the corresponding WAT constructs.

Usually the parser generates a parse tree, but this compiler generates an Abstract Syntax Tree, whose leaves are built from identifiers and constants (both are terminals). Each other node in the tree is associated to one of productions listed in *.y file.

At the implemented level, Bison performs shift operations and reduces (literally: stacking and reduction) during the analysis of productions; for each token in input that respects a rule, Bison will perform:

- Reduces terminal symbols performing the operations associated with the production
- The shift of non-terminal symbols performing the appropriate copy in the correct position in

the $$ pseudo-variable, passing the piece of code to the top-level node in the tree.
At the end of each production analysis, the $$ pseudo-variable will contain the whole AST.

In the last section, the epilogue, there are the definition of yyerror and the main function. Within the latter you call, after parsing is finished without syntax error, functions to print both AST and Symbol Table to standard output and save WAT code in appropriate *.wat file.

Grammar

In grammar, regarding function declarations, it has been planned that function can have parameters and return value only between void and basic data types such as int, float and char (strings are not supported because they are array of characters or, otherwise, pointer to characters). In addition, one or more parameters can only be passed by value and not by reference.
It has been implemented also rules for variable declarations (with or without initialization), function call, assignment, mathematical, logical and relational expression, input/output statements (with or without one or more variables passed, branching, iteration and return statements.

If statements can be nested or not and can be followed by else statement. The S/R conflict of the dangling else has been solved with the shift action so that the else-branch is attached to the nearby if-statement, as C language's convention.[2]

Initialization of for-loop statement cannot be empty but can have a comma-separated list of assignments.
Condition must be present and cannot have a comma-separated list of small conditions. However, many conditions can be evaluated using logical operators to bound them.
Increment statement cannot be empty but can have a comma-separated list of expressions.

[2] Examples of allowed syntax can be found in folder test/if_tests

Abstract Syntax Tree

In addition to syntax analysis, you need to give a meaning to every production and to verify the program input.

For an easy management and a fast implementation, has been decided to build and use an Abstract Syntax Tree using this particular scheme:

```
/* Define Abstract Syntax Tree structure */
struct AST {
    AstType type;
    union {
        AST_Const *ast_constant;
        AST_Variable *ast_variable;
        AST_Unary_Expr *ast_unary_expr;
        AST_Binary_Expr *ast_binary_expr;
        AST_Assign *ast_assign;
        AST_If_Stat *ast_if_stat;
        AST_For_Stat *ast_for_stat;
        AST_Return_Stat *ast_return_stat;
        AST_Builtin_Stat *ast_builtin_stat;
        AST_List *ast_list;
        AST_Def_Function *ast_def_function;
        AST_Call_Function *ast_call_function;
        AST_Body *ast_body;
        AST_Root *ast_root;
    };
};
```

Every specific AST node has its own attributes. For example, AST_Unary_Expr has two attributes:

```
typedef struct AST_Unary_Expr {
    UnaryExprType unary_type;       // increment, decrement, etc…
    AST *expression;
} AST_Unary_Expr;
```

Example

```
int sum(int a, int b) {
  int s = a + b;
  return s;
}

int main ()
{
  int c = 1, d = 2, somma;
  somma = sum(c,d);

  return 0;
}
```

After the parsing, you can see in terminal the following line:

```
Program GLOBAL DECLARATIONS
Program FUNCTIONS
    Function Definition NAME
        VARIABLE (INT type) sum
    Function Definition PARAMETERS
        List
            VARIABLE (INT type) a
            VARIABLE (INT type) b
    Function Definition BODY
        Function Body DECLARATIONS
            Assignment VARIABLE
                VARIABLE (INT type) s
            Assignment EXPRESSION
                BinaryExpr (Addition) LEFT
                    VARIABLE (INT type) a
                BinaryExpr (Addition) RIGHT
                    VARIABLE (INT type) b
        Function Body STATEMENTS
            Return_Statement
                VARIABLE (INT type) s
    Function Definition NAME
        VARIABLE (INT type) main
    Function Definition PARAMETERS

    Function Definition BODY
        Function Body DECLARATIONS
            Assignment VARIABLE
                VARIABLE (INT type) c
            Assignment EXPRESSION
                INT_CONSTANT 1
            Assignment VARIABLE
                VARIABLE (INT type) d
            Assignment EXPRESSION
                INT_CONSTANT 2
            VARIABLE (INT type) somma
        Function Body STATEMENTS
            Assignment VARIABLE
                VARIABLE (INT type) somma
            Assignment EXPRESSION
                Function Call NAME
                    VARIABLE (INT type) sum
                Function Call ARGUMENTS
                    VARIABLE (INT type) c
                    VARIABLE (INT type) d
            Return_Statement
                INT_CONSTANT 0
```

Before exiting, memory of each node of AST is released gradually, starting from inner nodes to outer nodes.

Semantic checks

Based on AST, it has been implemented some semantic checks. For declaration has been imposed that variables cannot have void type and array variable dimension must be constant. In addition, struct elements cannot be initialized during struct definition but only at struct variable declaration in the first part of source file.
While parsing, when a variable is found, compiler checks if it has already been declared. When a variable is declared, it can be initialized only by constants.

For function definition, parameters can be void or can be one or more basic data types.

In summary, some cases of incorrect semantics are:

- Use of a variable without having it first properly declared in the respective section of code;
- Inconsistency between data types in an assignment statement or in expression;
- Attempted redefinition of a variable;
- For function call checks if the function has been defined, if arguments are in the same number of parameters and if arguments and parameters have same type;
- For assignment statements checks if assignment variable and expression have the same type;
- For assignment statements, if variable on the left of assignment is a struct variable, checks only if it has been declared;
- For both I/O statements checks if variables passed have been declared;
- Verify that variable returned from a function is a simple variable and if correspond to a function return type in an assignment;
- For variable used as array index checks if has been previously declared and initialized, if it’s an integer variable and if it's a simple variable.

In case of error, compiler will display a fault message on the standard output, indicating the line number to which it occurred.

Symbol Table

A hierarchical symbol table has been designed starting from previous AST. This solution is the most suitable because it helped us to use the previous knowledge of the tree nodes structure to valorize the table itself.

So, the very basic symbol table is only composed of global variable and functions.

```
/* Symbol Table of whole program */
typedef struct SymTab {
    List *global_variables;         // list of SymTab_Variables
    List *functions;                // list of SymTab_Functions
} SymTab;
```

For individual variables and functions, a proper symbol table has been defined.

```
/* Symbol Table for all variables */
typedef struct SymTab_Variables {
    char *name;
    int n;                          // array dimension, -1 for simple variables and
-2 if we can't get array dimension
    ValType type;
    struct_info *s_info;            // !NULL only if variable is a struct
    int inizialized;
} SymTab_Variables;

/* Symbol Table for functions */
typedef struct SymTab_Functions {
    SymTab_Variables *func_name;
    List *parameters;               // list of SymTab_Variables
    List *local_variables;          // list of SymTab_Variables
} SymTab_Functions;
```

After execution on a test file, program will show the content of every symbol table generated.

Example

```
struct Point
{
int x;
int y;
} punto1 = {1,5}, punto2, punto3[2] = {{1,2},{3,4}};

int main ()
{
    struct Line
    {
        int a;
        int b;
    } line1, line2;

    punto2.x = 10;
    punto2.y = 20;

    punto1.x = punto1.y * 2;

    return 0;
}
```

```
GLOBAL SYMBOL TABLE
Name            Kind            Type            Dimension
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
punto1          VAR             STRUCT Point    /
punto2          VAR             STRUCT Point    /
punto3          VAR             STRUCT Point    2
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
STRUCT Point
x               VAR             INT             /
y               VAR             INT             /
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main            FUN             INT             /

MAIN SYMBOL TABLE
Name            Kind            Type            Dimension
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
line1           VAR             STRUCT Line     /
line2           VAR             STRUCT Line     /
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
STRUCT Line
a               VAR             INT             /
b               VAR             INT             /
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Issue

This symbol table, however, has one problem: if a variable is declared before of a struct and this variable has the same name of a variable inside the following struct, the program will generate a *segmentation fault*.
At the beginning, struct element variables are saved as global or local variables and only after, they are removed and stored as struct elements inside symbol table. That cause the segmentation error, because our code will delete the first variable with specified name found.
A possible limitation to this problem could be the following.

What there is now in grammar.y

declarations: declaration | *declarations declaration;*

With what could be replaced

declarations: struct_declarations var_declarations
struct_declarations: struct_declarations struct_declaration
var_declarations: var_declarations var_declaration

This will limit the issue but will not fully solve it. Declaration of two different struct with same variable name will generate anyway the segmentation fault.

Code generator

Code convertion is based on Abstract Syntax Tree and Symbol Table, allowing a one-to-one convertion.
Final code is obtained through a sequence of string concatenation and is saved as .wat file.
Then the .wat file is converted in a .wasm file, using the WABT's *wat2wasm* command.

At the end, compiler generate corresponding html, css and javascript files needed to run WebAssembly code[3]. In particular, javascript file creation is based on values of some flags, which enable import of memory and functions to WebAssembly module.

Issues

WebAssembly is a Minimum Viable Product (MVP), so some features are missing. Also, textual representation of WebAssembly code (.wat) seems to be very basic.

Main issues found are listed below:

- types allowed are limited (i32, i64, f32, f64)
- logical operators are not defined for float
- no direct support for strings and arrays
- no direct support for struct objects
- limited nondeterminism
- a single module can use at most 1 memory instance

At the beginning, Emscripten has been taken as reference to create textual representation, but it generates very big files (also with more than 10'000 code lines). Also, seems that Emscripten convert also used C libraries.[4]

For the implementation of this compiler, MDN WebAssembly documentation has been considered.

Array and Struct

To use more complex object like arrays and struct a memory management is strictly required. Unfortunately, this compiler works fine with strings, which are saved in a memory instance sequentially, but not with array or struct.

On the other hand, arrays and structs have not been considered at all. Arrays should be created outside WebAssembly code (in Javascript file) and passed through a memory instance with their length. This would increase the difficult of memory management, especially when used with strings.

Analog issue opened in WebAssembly Github repository.

[3] Full code for html, css and javascript can be found in doc/Site_Page folder

[4] For further information, see doc/Emscipten

Scanf conversion

There is no direct conversion of scanf instruction, due to limited nondeterminism. Web Assembly currently has no support for direct DOM and Browser API access and has to rely on JavaScript interop to update the DOM.

So, inputs must be asked in Javascript code and then passed to WebAssembly module as imported global mutable variable.

(global $v (import "js" "value") (mut i32))

This compiler, however, can't generate equivalent code for imported variables

Printf conversion

Also, printf instruction cannot be directly converted in textual WebAssembly code. There is a simple workaround, implemented in this compiler, that allows printing out integer, float, char and string, both variables and constants, on the html page.

Unfortunately, this case has a limitation too. An instruction like the following

printf("Value of v is %d", v);

would print to html page only variable's value. Assuming v is equal to 5, the html page will show 5.

Functions

list.h

```c
/* Create empty list */
List *list_new(void);

/* Return list length */
int list_length(List *list);

/* Add an item at the end of the list */
void list_append(List *list, void *item);

/* Remove an item in a specific position */
void list_remove(List *list, int index);

/* Get a specific element of the list */
void *list_get(List *list, int index);

/* Merge elements of two different lists */
List *list_merge(List *first_list, List *second_list);
```

ast.h

```c
/* ===== Create specific node ===== */
AST *new_AST_Const(ValType type, char *value);
AST *new_AST_Variable (char *name, int n, ValType type, struct_info *s_info, int
inizialized);
AST *new_AST_Unary_Expr (UnaryExprType unary_type, AST *expression);
AST *new_AST_Binary_Expr (BinaryExprType binary_type, AST *right, AST *left);
AST *new_AST_Assign (AST *variable, AST *expression);
AST *new_AST_If_Stat (AST *condition, AST *then_branch, AST *else_branch);
AST *new_AST_For_Stat (List *init, AST *condition, List *increment, AST *loop);
AST *new_AST_Return_Stat (AST *expression);
AST *new_AST_Builtin_Stat (BuiltinFunction function, AST *content, List
*variables);
AST *new_AST_List (List *list);
AST *new_AST_Def_Function (AST *func_name);
AST *new_AST_Call_Function (AST *func_name, List *arguments);
AST *new_AST_Body (List *declarations, List *statements);
AST *new_AST_Root (List *global_declaration, List *functions);

/* ===== Functions to print AST nodes ===== */
char *ast_type_name(AST *ast);
void print_ast(AST *ast, int indent);

/* ===== Functions to free AST nodes ===== */
void free_ast_list(List *ast_list);
void free_ast(AST *ast);
```

symtab.h

```
/* ===== Functions to create Symbol Table ===== */
SymTab *init_symtab();
SymTab_Functions *new_SymTab_Functions (SymTab_Variables *func_name);
SymTab_Variables *new_SymTab_Variables (char *name, int n, ValType type,
struct_info *s_info, int inizialized);
struct_info *new_struct_info (char *name, List *elements);

/* ===== Function to update Symbol Table ===== */
void insert_fun(SymTab *symtab, SymTab_Functions *sym_fun);
void insert_var(SymTab *symtab, SymTab_Variables *sym_var, char *scope);
void remove_symtab_variable(SymTab *symtab, char *scope, int pos);
void update_par(SymTab *symtab, List *parameters, char *scope);

/* ===== Functions to query Symbol Table ===== */
int lookup (SymTab *symtab, char *name, char *scope, int *where);
SymTab_Variables *get_symtab_var(SymTab *symtab, char *scope, int pos, int where);
void check_redeclaration(SymTab *symtab, char *name, char *scope);

/* ===== Functions to print Symbol Table ===== */
void print_symtab(SymTab *symtab);
void print_symfun(SymTab_Functions* symfun);
void print_symvar(SymTab_Variables *symvar,char *kind, List *struct_infos);
void print_struct_info(List *struct_infos);
void print_tilde();                                 // auxiliary
```

wat.h

```
/* ===== Function to generate .wat file ===== */
void code_generation(AST *ast, SymTab *symtab, char *filename);
char *indentation (int n);
char *convert_code (AST *ast, SymTab *symtab, char *scope, int indent);

/* ===== Create auxiliary files ===== */
void createAuxFiles(char *filename);
void wat2wasm(char *filename);
void createHTML();
void createCSS();
void createJS(char *filename);
```

utils.h

```
/* ===== Utils functions ===== */
/*  Concatenation of many strings  */
char* concat(int n_token, char *sep, char *token,...);     /* concat tokens of a
rule */
/*  Convert list of AST into list of SymTab_Variables  */
List *convert(List *start);
/*  Prepare List *struct_element needed to create struct_info  */
List *prepare_struct_elements(int n, List *elements);

/* ===== Sematic functions ===== */
/*  Update node type in AST using Symbol Table  */
void update_node_type(AST *node, SymTab *symtab, int where, int pos);
/*  Update inizialization flag of a variable in Symbol Table  */
void update_inizialization(SymTab *symtab, char *name, char *scope);
/*  Update inizialization flag of a struct element variable in Symbol Table  */
void update_struct_element_init(SymTab *symtab, char *name, char *scope, int
array_list, int pos_elem);
/*  Evaluate type of expression rule  */
ValType evaluate_expression_type(AST *ast, SymTab *symtab, char *scope);
/*  Check type of inizialization list elements  */
ValType check_array_init_list(char *var_assign_name, List *list);
/*  Semantic check on variable assignment  */
int check_decl_assignment(SymTab_Variables *var_assign, AST *expression, SymTab
*symtab, char *scope);
/*  Check matching between arguments and parameters  */
void check_args_params(SymTab *symtab, char *scope, char *function_name, List
*args);
/*  Semantic check on Assignment Node  */
int check_assignment(AST *ast_assignment, SymTab *symtab, char *scope);
/*  Check if variable is found in Symbol Table  */
void is_var_declared (AST *ast, SymTab *symtab, char *scope);
/*  Verify return identifier's type  */
void verify_return_id_type (AST *ast, SymTab *symtab, char *scope);
/*  Verify return statement  */
void check_return(ValType func_type,AST *body, SymTab *symtab, char *scope);
```

References

Bison Manual
Book: Flex and Bison - John Levine
C operator precedence
WebAssembly Semantics
WebAssembly - Text format specification
WebAssembly - Future features
MDN - Understanding the text format
MDN - Using the JavaScript API
MDN - Array example (html+js) - MDN - Array example (wat)