AdvCalc++ Trans-compiler

Yusuf AKIN & Atakan YASAR

2023-04-30

Contents

1 AdvCalc++ Trans-compiler **2**
- 1.1 Build and Execution . . . 2
- 1.2 AdvCalc++ Language Specification . . . 2
  - 1.2.1 Variables and Constants . . . 2
  - 1.2.2 Operators . . . 2
  - 1.2.3 Expressions . . . 3
  - 1.2.4 Assignment Statement . . . 4
  - 1.2.5 Comments . . . 4
- 1.3 Program Structure and Implementation . . . 4
  - 1.3.1 main.c . . . 4
  - 1.3.2 lexer.c . . . 5
  - 1.3.3 parser.c . . . 8
  - 1.3.4 transpiler.c . . . 11
  - 1.3.5 dictionary.c . . . 13
  - 1.3.6 Example Module Outputs . . . 15

2 FAQ **17**

1 AdvCalc++ Trans-compiler

AdvCalc2ir is a trans-compiler that translates **AdvCalc++** source code to **LLVM IR** code. It is developed as a part of the **CmpE230: Introduction to Systems Programming** course at **Bogazici University**.

AdvCalc++ is a simple calculator language that supports basic arithmetic and bitwise operations. Apart from the added support for modulo and division operations, **AdvCalc++** is a backwards compatible extension of the **AdvCalc** language. There are also some bug fixes.

1.1 Build and Execution

To build, ensure **gcc** is installed in your machine. Project has no dependencies other than the standard c library.

To compile the program from the source, run makefile from the top level directory:

```
make
```

The executable produced is named **"advcalc2ir"**, to translate a source file to LLVM IR, run:

```
./advcalc2ir <advcalc++_source_file>.adv
```

The output file will be named **"<advcalc++_source_file>.ll"**. To compile the LLVM IR code to an executable, run:

```
clang <advcalc++_source_file>.ll -o <advcalc++_source_file>
```

or if you want to use **lli** to execute the code:

```
lli <advcalc++_source_file>.ll
```

1.2 AdvCalc++ Language Specification

1.2.1 Variables and Constants

All the variables and constants are assumed to be 32-bit integers. Notice this is different from the original AdvCalc language, where variables and constants were 64-bit integers.

- Variable names are case-sensitive.
- Variable names can contain only uppercase and lowercase letters.
- Undeclared variables cannot be used.
- Reserved words cannot be used as variable names (e.g. xor, not, ls, rs, rr, lr).

1.2.2 Operators

All the operations listed below work with **64-bit** integers.

1.2.2.1 Binary Arithmetic Operations AdvCalc++ supports the following binary infix arithmetic operations:

Operation	Use	Description
$+$	$a + b$	Sum of a and b
$-$	$a - b$	Difference of a and b
$*$	$a * b$	Product of a and b
$/$	a/b	Quotient of a and b
%	$a\%b$	Remainder of a divided by b

1.2.2.2 Binary Bitwise Operations AdvCalc++ supports the following binary infix bitwise operations:

Operation	Use	Description
&	$a \ \& \ b$	Bitwise AND of a and b
|	$a \mid b$	Bitwise OR of a and b

1.2.2.3 Unary Bitwise Operations The only unary bitwise operation supported is the bitwise NOT operation, it is used as a prefix operator (i.e. a function)):

Operation	Use	Description
$not(.)$	$not(a)$	Bitwise complement of a

1.2.2.4 Other Bitwise Operations Following functions are supported for bitwise operations:

Operation	Use	Description
$xor(.,.)$	$xor(a,b)$	Bitwise XOR of a and b
$ls(.,.)$	$ls(a,b)$	Bitwise left shift of a by b bits
$rs(.,.)$	$rs(a,b)$	Bitwise right shift of a by b bits
$rr(.,.)$	$rr(a,b)$	Bitwise right rotation of a by b bits
$rl(.,.)$	$rl(a,b)$	Bitwise left rotation of a by b bits

1.2.3 Expressions

Expressions are evaluated in the following order, from highest to lowest precedence:

1. Parenthesized expressions (i.e. expressions enclosed in "(" and ")"), and function calls.
2. Multiplication ("*"), division ("/"), and modulo ("%") operations are left associative (i.e. $a * b/c$ is evaluated as $(a * b)/c$, not $a * (b/c)$).
3. Addition ("+") and subtraction ("-"). The operations are left associative (i.e. $a-b+c$ is evaluated as $(a - b) + c$, not $a - (b + c)$).

4. Bitwise and ("&"))
5. Bitwise or ("|")

1.2.4 Assignment Statement

Assignment is done using the "=" operator. The left-hand side of the assignment must be a variable name. The right-hand side can be any valid expression. - There is no explicit variable declaration syntax. - Variable declaration and assignment is simultaneous. - It is illegal to use a variable before it is declared.

```
> a = 5
> b = 10 + c % error since c is not declared
> a + b
15
```

1.2.5 Comments

Contrary to **AdvCalc** language with single line comments, comments are not supported in **AdvCalc++**. This is due to the clash of the "%" character with the modulo operator. But it is easy to extend the lexer to support comments. See the **lexer.c** line 170 for the implementation of the comment support.

1.3 Program Structure and Implementation

AdvCalc++ is implemented in **c**. Since **AdvCalc++** is an interpreted language, it is possible to _translate the source code line by line. In fact, the program reads the source code line by line, tokenizes it, parses it, and generates the corresponding **LLVM IR** code.

The program is divided into the following modules:

- **main.c**: The main module of the program. All io operations are done within this module.
- **lexer.c**: The lexer module. It contains the functions that tokenize the input string.
- **parser.c**: The parser module. It contains the functions that parse the input string and convert it to a parse tree.
- **transpiler.c**: The trans-compiler module. It contains the functions that generate the LLVM IR code.
- **dictionary.c**: The dictionary module. Implements a lookup table to ensure that all the variables are declared before use.

1.3.1 main.c

This module reads in a file with the name given as a command-line argument, transpiles it, and outputs the LLVM IR to a new file with the same name but a ".ll" extension. The program includes several functions that are used to prelog, which writes some boilerplate code at the beginning of the output file (implementations of the functions **lr** and **rl** is included

in prelog); epilog, which writes some code at the end of the output file; and main, which is the main function of the program.

The program begins by checking that the user has supplied a filename as a command-line argument. If there is no argument, an error message is displayed and the program exits. If a filename is supplied, the program attempts to open the file for reading. If it is unable to open the file, an error message is displayed and the program exits.

The program then creates a new output file with the same name as the input file but with a ".ll" extension. If it is unable to create the output file, an error message is displayed. The prelog function is then called to write some boilerplate code at the beginning of the output file.

The program then reads the input file line by line and passes each line to the translation function. The **"translate"** function is responsible for transpiling the line of code and writing the LLVM IR to the output file. If the **"translate"** function returns a state of 2, an error message is displayed and the program sets an error flag. The error message shows the line number of the error in the source code.

After all lines have been processed, the program calls the epilog function to write some code at the end of the output file. The program then closes both the input and output files.

If the error flag has been set, indicating that an error occurred during the trans compilation process, the output file is removed. If no errors occurred, the program exits with a status of 0.

WARNING: The program doen not check for any file extension. The program will overwrite any file with the same name and a ".ll" extension. And in case of an error, the program will remove the output file, so, if you have a file with the same name and a ".ll" extension, it will be removed.

1.3.2 lexer.c

```c
#define MAX_TOKEN_LEN 257
/* Token types */

typedef enum TokenType {

  TOKEN_UNKNOWN,    // 0
  TOKEN_LITERAL,    // 1
  TOKEN_IDENTIFIER, // 2

  TOKEN_STAR,  // 3
  TOKEN_PLUS,  // 4
  TOKEN_MINUS, // 5

  TOKEN_AND, // 6
```

```
  TOKEN_OR,  // 7
  TOKEN_NOT, // 8
  TOKEN_XOR, // 9

  TOKEN_LS, // 10
  TOKEN_RS, // 11
  TOKEN_LR, // 12
  TOKEN_RR, // 13

  TOKEN_EQ,      // 14
  TOKEN_COMMENT, // 15
  TOKEN_EOF,     // 16

  TOKEN_LEFT_PAREN,  // 17
  TOKEN_RIGHT_PAREN, // 18
  TOKEN_COMMA,       // 19
  TOKEN_MOD,         // 20
  TOKEN_DIV,         // 21
} TokenType;


/* Token struct
 * type: type of token
 * start: start location of token
 * length: length of token
 * value: value of token if it is a literal
 */
typedef struct Token Token;
struct Token {
  TokenType type;
  char *start;
  size_t length;
  long value;
};

/* Lexer struct
 * input: input string
 * cur_pos: current position in input string
 * input_len: length of input string
 * token_list: list of tokens
 * cur_token: current token
 */
typedef struct Lexer Lexer;
struct Lexer {
```

```
  char *input;
  size_t cur_pos;
  size_t input_len;

  Token token_list[MAX_TOKEN_LEN];
  size_t cur_token;
};


/* Create a new lexer
 * input: input string
 * input_len: length of input string
 */
Lexer *lexer_new(char *input, size_t input_len);


/* Free a lexer
 * lexer: lexer to free
 */
void lexer_free(Lexer *lexer);

/* Prints the tokinized input
 * lexer: lexer to print
 */
void print_lex(Lexer *lexer);

/* Get the next token
 * lexer: lexer to get token from
 */
void lexer_next(Lexer *lexer);
```

The lexer reads in a string of characters and tokenizes it into a list of tokens, where each token represents a symbol or word in the programming language.

The lexer_next function starts by skipping over any whitespace characters in the input string. It then examines the current character to determine what type of token it represents. For example, if the current character is a plus sign, it creates a new token with type TOKEN_PLUS. If the current character is a left parenthesis, it creates a new token with type TOKEN_LEFT_PAREN, and so on.

If the current character is not one of the recognized symbols, the lexer tries to parse it as a number. If the number is valid, it creates a new token with type TOKEN_LITERAL and the parsed value. If the number is not valid, it creates a new token with type TOKEN_UNKNOWN.

If the end of the input string is reached, the lexer creates a new token with type TO-

KEN_EOF.

The tokens produced by the lexer are stored in an array of Token objects, which is a member of the lexer object. The lexer_next function updates the current token index of the lexer object each time it produces a new token.

1.3.3 parser.c

```c
typedef enum SyntaxNodeType { BINOP, UNOP, PAREN, TOKEN, ERROR } SyntaxNodeType;

typedef struct SyntaxNode SyntaxNode;

/*
 * SyntaxNode Struct
 * type: type of the node,
 *       BINOP: left child, mid child as binary operator, right child,
 *       UNOP: left child, mid child as unary operator,
 *       PAREN: left child as inside of parenthesis,
 *       TOKEN: token type for leave
 * left: left term of the tree
 * mid: contains operators
 * right: right term of the tree
 * token: leave of the tree, tokens are stored here
 */
struct SyntaxNode {
  SyntaxNodeType type;
  SyntaxNode *left;
  SyntaxNode *mid;
  SyntaxNode *right;
  Token *token;
};

typedef struct SyntaxTree SyntaxTree;

struct SyntaxTree {
  SyntaxNode *root;
};

/*
 * generates a parse tree
 * calls parse_expr
 * input: Token array
 */
SyntaxTree *parse(Token *tokens);
```

```
/*
 * <expr> = <term><eof> | <eof>
 * calls parse_term then expects EOF token
 * input: current token
 */
SyntaxNode *parse_expr(Token **tokens);

/*
 * parses <term>
 * calls and returns parse_comma
 */
SyntaxNode *parse_term(Token **tokens);

/*
 * parses <term>,<term>
 * calls parse_or and assigns it to left child as left <term>
 * then if comma exists
 * assigns TOKEN_COMMA to mid child
 * calls parse_comma and assigns it to right child as right <term>
 */
SyntaxNode *parse_comma(Token **tokens);

/*
 * parses <term>|<term>
 * calls parse_and and assigns it to left child as left <term>
 * then if or exists
 * assigns TOKEN_OR to mid child
 * calls parse_and and assigns it to right child as right <term>
 */
SyntaxNode *parse_or(Token **tokens);

/*
 * parses <term>&<term>
 * calls parse_plus_minus and assigns it to left child as left <term>
 * then if and exists
 * assigns TOKEN_AND to mid child
 * calls parse_plus_minus and assigns it to right child as right <term>
 */
SyntaxNode *parse_and(Token **tokens);

/*
 * parses <term>(+|-)<term>
 * calls parse_mul and assigns it to left child as left <term>
 * then if plus or minus exists
```

```
 * assigns TOKEN_PLUS or TOKEN_MINUS to mid child
 * calls parse_mul and assigns it to right child as right <term>
 */
SyntaxNode *parse_plus_minus(Token **tokens);

/*
 * parses <term>*<term>
 * calls parse_primary and assigns it to left child as left <term>
 * then if star exists
 * assigns TOKEN_STAR to mid child
 * calls parse_primary and assigns it to right child as right <term>
 */
SyntaxNode *parse_mul(Token **tokens);

/*
 * parses <primary>
 * calls and return parse_func
 */
SyntaxNode *parse_primary(Token **tokens);

/*
 * parses <funcname>(<term>,<term>)
 * first token must be valid <funcname>
 * then calls parse_paren
 * then if comma exits in parenthesis
 * assigns left <term> to left child
 * assigns related token to mid child
 * assigns right <term> to right child
 */
SyntaxNode *parse_func(Token **tokens);

/*
 * parses <var>
 * expects TOKEN_IDENTIFIER or TOKEN_LITERAL
 */
SyntaxNode *parse_var(Token **tokens);

/*
 * parses (<term>)
 * expects TOKEN_LEFT_PAREN as first token
 * calls parse_term
 * at the end expects TOKEN_RIGHT_PAREN as last token
 */
SyntaxNode *parse_paren(Token **tokens);
```

```c
/*
 * Free all syntax nodes recursively
 */
void free_syntax_tree(SyntaxTree *tree);

/*
 * prints syntax nodes indented according to depths
 */
void print_syntax_tree(SyntaxTree *tree);
```

”parser.c” implements the parser module. The output is a parse tree. Used grammar:

```
<expr> = <term><eof> | <eof>
<term> = <term><binop><term> | <primary> | (<term>)
<primary> = <func> | <var>
<func> = <funcname>(<term>,<term>) | <unop>(<term>)
<var> = literal or identifier token
<binop> = [*+-&|\\%]
<funcname> = (xor|ls|rs|lr|rr)
<unop> = not
```

This is a recursive descent parser implementation that uses a top-down approach for parsing expressions. The code reads in tokens from the input and builds a syntax tree that represents the structure of the input expression.

The main parsing function is parse_expr, which calls parse_term to parse terms, which in turn calls parse_comma to parse comma-separated lists of terms, and so on down the line, with each function handling a specific level of operator precedence.

For example, parse_or handles the logical OR operator, while parse_and handles the logical AND operator. These functions use a while loop to continue parsing until no more operators of that precedence level are found.

The parse_expr function is called from the parse function, which is the entry point for the parser module.

For more details, see the **AdvCalc** documentation.

1.3.4 transpiler.c

```c
/*
/*
 * Translates the given expression to LLVM IR and writes it to the given file.
 *
 * Returns following values, depending on the result:
 * 0: expression
 * 1: assignment
```

```
 * 2: error
 * 3: empty line
 *
 * dict: dictionary storing the declared variables
 *
 * The function creates its own lexer and parser, and uses them to parse the input str
 */
int translate(Dictionary *dict, char *input, FILE* output_file);

/*
 * Returns a new register number to use for temporary storage.
 */
int new_register();

/*
 * Loads the value of the given variable into a register, and writes the corresponding
 * LLVM IR to the output file.
 *
 * Returns 0 if successful, or 1 if an error occurs.
 *
 * dict: dictionary storing the declared variables
 * value: name of the variable to load
 * output_file: file to write LLVM IR to
 */
int load(Dictionary *dict, char *value, FILE *output_file);

/*
 * Declares the given variable, and writes the corresponding LLVM IR to the output fil
 *
 * Returns 0 if successful, or 1 if an error occurs.
 *
 * dict: dictionary storing the declared variables
 * variable: name of the variable to declare
 * output_file: file to write LLVM IR to
 */
int declare(Dictionary *dict, char *variable, FILE *output_file);

/*
 * Writes the value of the given register to the output file.
 *
 * output_file: file to write LLVM IR to
 * reg: token representing the register to print
 */
void print_var(FILE *output_file, Token* reg);
```

```c
/*
 * Writes the LLVM IR for the given binary operator to the output file, using the give
 * left and right operand registers.
 *
 * Returns a pointer to the register containing the result of the operation.
 *
 * type: type of the binary operator
 * root: root token of the expression subtree
 * left_reg: register containing the left operand
 * right_reg: register containing the right operand
 * output_file: file to write LLVM IR to
 * error: pointer to boolean flag indicating whether an error occurred
 */
Token *print_op(enum SyntaxNodeType type, Token *root, Token *left_reg, Token *right_reg


/*
 * Stores the value of the given register in the given variable, and writes the corres
 * LLVM IR to the output file.
 *
 * name: name of the variable to store the value in
 * reg: token representing the register containing the value to store
 * output_file: file to write LLVM IR to
 */
void store(Dictionary *dict, char *name, Token *reg, FILE *output_file);
```

This module implements transpiler from the AdvCalc++ language to LLVM IR. The transpiler takes as input a syntax tree generated by parsing an **AdvCalc++** program and outputs LLVM IR code that can be compiled into executable code by a compiler like LLVM.

The transpiler code consists of several functions that perform different tasks. The **"translate"** function is the main function that takes the syntax tree as input and outputs LLVM IR code.

The basic idea is to recursively traverse the syntax tree and generate LLVM IR code for each node. The **"translate"** function calls the **"translate_expr"** function to translate the root node of the syntax tree. The **"translate_expr"** function then calls the **"translate_expr"** function to translate the left and right child nodes of the root node. This process continues recursively until the entire syntax tree has been translated.

1.3.5 dictionary.c

This module implements a hash table to store the variables. The hash table handles collisions using separate chaining.

```c
#define HASHSIZE 31013
#include <stdint.h>
typedef int32_t operand_t;
/*
 * Chain struct
 * next: pointer to next entry in chain
 * name: name of variable
 * value: value of variable
 */
typedef struct Chain Chain;
typedef struct Chain { /* table entry: */
    Chain *next; /* next entry in chain */
    char *name; /* defined name */
    operand_t value;
}Chain;

/*
 * Dictionary struct
 * hashtab: array of pointers to chains
 */
typedef struct Dictionary Dictionary;

struct Dictionary{
  Chain* hashtab[HASHSIZE];
};

/*
 * Create a new dictionary
 */
Dictionary* new_dictionary();

/*
 * Set a variable in the dictionary
 * dict: dictionary to set variable in
 * name: name of variable
 * value: value of variable
 */
void declare_var(Dictionary *dict, char *name);

/*
 * Get a variable from the dictionary
 * dict: dictionary to get variable from
 * name: name of variable
 */
```

```
int is_declared(Dictionary *dict, char *name);

/*
 * Free a dictionary
 * dict: dictionary to free
 */
void free_dict(Dictionary* dict);

/*
 * Print a dictionary
 * dict: dictionary to print
 */
void print_dict(Dictionary* dict);

/*
 * Hash function
 * s: string to hash
 */
unsigned hash(char *s);
```

This module implements a hash table for storing the information of whether variables are declared or not. The program includes functions for declaring a variable in the hash table, and checking if a variable has been declared.

The implementation uses separate chaining for handling collisions. Each bucket in the hash table is a linked list of chains. Each chain contains the name of a variable and a flag indicating whether the variable has been declared or not.

The "declare_var" function takes a Dictionary and a string as input, and declares the variable by adding it to the hash table using the "put" function.

The "is_declared" function takes a Dictionary and a string as input, and checks if the variable has been declared by looking it up in the hash table using the "get" function. If the variable is found, it has been declared and 1 is returned. If the variable is not found, it has not been declared and 0 is returned.

1.3.6 Example Module Outputs

Input **AdvCalc++** program:

```
// test1.adv
x=3
y=5
zvalue=23+x*(1+y)
zvalue
k=x-y-zvalue
```

```
k=x+3*y*(1*(2+5))
k+1
```

Traslating...

```
$ ./advcalc2ir test1.adv
$ ls
test1.adv test1.ll
```

Output **LLVM IR** code:

```
; ModuleID = 'advcalc2ir'

declare i32 @printf(i8*, ...)
@print.str = constant [4 x i8] c"%d\0A\00"

define i32 @lr(i32 %x, i32 %c) {
  %l1 = shl i32 %x, %c
  %l2 = sub i32 32, %c
  %l3 = lshr i32 %x, %l2
  %l4 = or i32 %l1, %l3
  ret i32 %l4
}
define i32 @rr(i32 %x, i32 %c) {
  %l1 = lshr i32 %x, %c
  %l2 = sub i32 32, %c
  %l3 = shl i32 %x, %l2
  %l4 = or i32 %l1, %l3
  ret i32 %l4
}
define i32 @not(i32 %x) {
  %n1 = xor i32 %x, -1
  ret i32 %n1
}

define i32 @main() {
  %x = alloca i32
  store i32 3, i32* %x
  %y = alloca i32
  store i32 5, i32* %y
  %zvalue = alloca i32
  %reg_1 = load i32, i32* %x
  %reg_2 = load i32, i32* %y
  %reg_3 = add i32 1, %reg_2
  %reg_4 = mul i32 %reg_1, %reg_3
  %reg_5 = add i32 23, %reg_4
```

```
  store i32 %reg_5, i32* %zvalue
  %reg_6 = load i32, i32* %zvalue
  call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0,
  %k = alloca i32
  %reg_7 = load i32, i32* %x
  %reg_8 = load i32, i32* %y
  %reg_9 = sub i32 %reg_7, %reg_8
  %reg_10 = load i32, i32* %zvalue
  %reg_11 = sub i32 %reg_9, %reg_10
  store i32 %reg_11, i32* %k
  %reg_12 = load i32, i32* %x
  %reg_13 = load i32, i32* %y
  %reg_14 = mul i32 3, %reg_13
  %reg_15 = add i32 2, 5
  %reg_16 = mul i32 1, %reg_15
  %reg_17 = mul i32 %reg_14, %reg_16
  %reg_18 = add i32 %reg_12, %reg_17
  store i32 %reg_18, i32* %k
  %reg_19 = load i32, i32* %k
  %reg_20 = add i32 %reg_19, 1
  call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8], [4 x i8]* @print.str, i32 0,
  ret i32 0
}
```

File including error.

```
// test2.adv
x=3
y=5
zvalue=23+x*(1+undeclaredVariable)
zvalue
k=x-y
```

```
$ ./advcalc2ir test2.adv
$ ls
test2.adv
```

Standard output:

```
Error on line 3!
Error on line 4!
```

2 FAQ

Q: Any improvements or extensions? - Yes, we can add more operators, functions, and variables.

Q: How does the parser work?

- See LL(1) Grammars, and Recursive Descent Parser. This is basically how the parser works, we didn't reinvent the wheel.

Q: What difficulties did you encounter? - Since we have already implemented the parser implemented for **AdvCalc**, translating the syntax tree to LLVM IR was a relatively straightforward process.

Q: What did you learn from this project? - We learned how to use LLVM IR to generate executable code from a high-level language.

Q: What would you do differently if you were to do this project again? - We would add more make the parser more extensible by adding more operators, functions, and variables.

Q: How to get a pdf version of the README.md ?

- Easy,

```
pandoc --pdf-engine=xelatex \
    --highlight-style tango \
    -V colorlinks \
    -V urlcolor=NavyBlue \
    -V toccolor=red \
    --toc -N \
    -V geometry:margin=1in \
    -V fontsize:12pt \
    -s -o Documentation.pdf README.md
```