Contents

DPCC: DParo’s Own C-Alike Compiler

Davide Paro

December 2020

1 Assignment Description

This project is about an assignment for a course on **Compilers** at the department of Computer Engineering Master Degree Padova (ITA).

The assignment consists in implementing a toy compiler for a toy language. More emphasis is put on the implementation for the frontend side (input, lexing, parsing, type checking), while the backend side is stubbed out by a simple 3AC [1] Intermediate Code generator. We are free to design the syntax of this toy language however we like.

The assignment specs out the how the compiler should be composed. We can in fact distinguish these macro components:

- **Input Stage** deals with the input byte stream that composes the source of the program.
- **Lexer/Scanner** has the purpose of grouping characters (lexical analysis) together to compose compunded structures (called tokens). For the project assignment we can use **Flex** to aid in the implementation of the scanner.
- **Parser** for performing the syntax analysis. It is what defines the look & fell (grammar) of the language. For the project assignment we can use **Bison** to aid in the implementation of an LR parser.
- **Intermediate Code Generator**. The ultimate purpose of a compiler is to produce something useful. In this project assignment we are not asked to implement a proper backend. Instead, we need to emit a 3AC representation of our input program. More in this later.

Three Adress Code (3AC) is a type of intermediate code generator which is both easy to understand and to emit. In 3AC, each statement can only have 1 operand at the left hand side of the assignment, and at most 2 operands at the right hand side of the assignment, and an operator driving the operation that should be performed.

You can view more about 3AC at the following Wikipedia link, or here Geeks4Geeks link.

In practice, the emitted 3AC code is on itself a partially valid C program, it’s only missing variable declarations at the top for each temporary variable. Control flow is allowed to be implemented trough the usage of C labels and simple if conditional followed by a goto statement. Inside the if conditional there can only be a single element composing the expression.

The project assignment requires the following features that should be developed for the language:

- Variables declaration, initialization and assignment
- Scoping. Variable names are reusable in different scopes. Variable shadowing may or may not warn/fail/pass depending on the design choices.
- Only 2 types of variables: integers, booleans
- Assignment statements, print statements, if statements, and at least 1 loop statement at our liking
- Handling of simple mathematical expressions that we can encounter in common programming languages: addition, subtraction, multiplication, division, modulo, etc ...
- **Function definition, function calls, and custom user definable types are not required**

The unpatient reader can already jump tho the Appendix A to quickly look at an example program of the designed language. Appendix A lists the program source, the output which is generated by running said program and the 3AC that the compiler emits when performing the compilation.

---

[1] 3AC: Three Address Code

2 DPL and dpcc: A quick peak at the language and at the compiler

DPL and **DPCC** are respectively the **name of the language** and the **name of the implemented compiler**. They are named after their author.

From now on **DPL** and **DPCC** will be used for brevity for refering to the language and to the compiler. We will use **dpcc** (all lowercase) instead, to refer to the actual executable where the compiler lives.

That being said **DPCC** (all UPPERCASE) and **dpcc** (all lowercase) are mostly used interchangeably to refer to the same thing.

2.1 DPL: Structure of the language

DPL is mostly a C-alike compatible language, in fact it borrows most of its syntax and semantics. **DPL** is also inspired by the following programming languag: **Rust** & **JS** especially in the syntax used to perform variable declarations (usage of the keyword **let**).

Rust is a modern system programming language with strong typing guarantees. Among all the interesting features that Rust provides one of them is type deduction. Thanks to Rust's strong typing guarantees and thanks to strong type deduction rules implemented inside the **rustc** compiler, Rust allows one to declare variables with a very low-weight syntax, similar to a syntax provided from a typical dynamic language (for example JS).

DPL, like Rust, also have a very simple form of a type deduction system. It is not even remotely close to the Rust type deduction system, but still it allows the user of the language to not always need to specify the type of each variable in a declaration. How this is achieved will be described in later sections.

In **DPL**:

- Spaces and newlines mostly do not matter, they simply introduce token boundaries.
- Comments start with the double forward slash '//', and C-style multiline comments are instead not supported (at the time of writing).

Here follows a chunk of **DPL** code to show off the syntax and some of the language features:

```
// Print statement with immediate C-style strings. C-style strings can only be used inside print statement
print("Hello world\n");

// Variable declaration and initialization
let a = 10;        // Integer Type deduced
let f = 10.0;      // Float type deduced
let b = false;     // Boolean type deduced

// Explicit types
let i: int = 0xffff & ~0xb00111;
let f: float = 10.0 + 20.0 ** 2;

// Immediate values can be printed
print(10);
print(30 + 4);

// Variables can be printed
print(i);      // Print integer
print(f);      // Print float

// Casting can be used to enforce type conversion
let myInt: int = int(10.00f);
let myFloat = float(0xFF);

// Type deduction
let b = (10 < 20); // Boolean type is deduced
let f = 1 + 2.0f;  // Float type is deduced (the 1 is upcasted to a float)

// Scoping and restricting variable declarations to the current scope
{
    // Simple single dimension arrays declaration
    let buf_i: int[100];        // Known size integer array
    let buf_f: float[100];      // Known size float array

    // Integer array with deduced size from the RHS initializer list
    let buf = [ 10, 20, 30, 40, 50 ];
    let fs: float[] = [ 0.1, 0.2, 0.3, 0.4, 0.5 ];

```

```
    // Arrays can be printed
    print(buf);
}

let buf: int[100];

// Control flow
for (let i = 0; i < 100; i++) {
    buf[i] = i ** 2;

    if (buf[i] == 10) {
        print("buf[i] is 10!!!\n")
    }
    else if (buf[i] == 20) {
        print("buf[i] is 20!!!\n");
    }
    else {
        print("None of above\n");
    }
}
```

The cool thing about **DPL**, is that it is **almost a Javascript subset**. That is one can simply copy the **DPL** code, strip the type information (if they're used anywhere) by manual editing or automatically, and paste the same code into a browser console to evaluate as JS code. With a couple modifications here and there (for example arrays with no initalizer list must be converted into a valid JS array), one can test if the compiler **dpcc** is producing the correct output by simply evaluating the same code in a browser console.

This example shows how to convert **DPL** into **JS** by manual editing:

```
// This is a chunk of DPL code
let a = 10;                 // This is also valid JS code
let b = [ 10, 20, 30, 40 ]; // This is also valid JS code
let c: int = 10;            // Mostly valid JS code, remove the type info and the colon
let d: int[1024];           // Js arrays grow on demand automatically when touching elements.
                            // No need to specify neither type nor number of elements

print(d);                   // Valid JS code if function print were to be defined
```

Here's the equivalent JS code:

```
// This is the equivalent Javascript
const print = console.log;  // Define it once at the top of the script

let a = 10;                 // Same as before
let b = [ 10, 20, 30, 40 ]; // Same as before
let c = 10;                 // Just strip the int type
let d = [];                 // Just initalizing with an empty array is enough

print(d);                   // Works because print is defined at the top
```

Most other **DPL** syntax and features, like code blocks, conditionals, loops . . . etc are valid JS code thanks on how the grammar for **DPL** was defined.

For now **dpcc** supports only 5 types: `bool`, `int`, `float`, `string`, `bool[]`, `int[]`, `float[]`. Only single dimensions array are for now supported. So arrays do not generalize to any number of dimensions.

Most of these types have full on semantics, meaning that the compiler can deduce a type of an expression given the types of its operand. In some cases it can reject the code if the operands of an expression have invalid types. At the current time of writing, `string` types are quirky, meaning that they don't have a full type tracking inside the compiler like other types do. The compiler still knows what a string is, and in it marks correctly **string literals** as a `string` type, but strings undergo different semantics. They cannot be assigned or operated on like a variable, but instead, they can only be used as a parameter to the print statement.

2.2 DPCC: Using the compiler

The **dpcc** compiler is written in the **C** language. Unfortunately at the current time of writing **dpcc** works only under Unix like operating systems. The compiler has been tested under **Ubuntu 20.10**, Ubuntu 20.04, and macOS 10.15. The compiler was developed by his author using an **Ubuntu 20.10** machine, while the other distro/OS were tested thanks to *Github Actions* automated build-check cycles. Windows builds failed due to MSVC rejecting the source code of **dpcc** cause it contains some GCC extensions and some hard coded unix syscalls. In short words **dpcc** can be only compiled with either GCC or CLANG compilers and executed in a Unix/Posix compatible operating system.

If you would like to build the compiler yourself from scratch please refer to the Project WIKI[2]

The compiler can and should be invoked from the commandline. The **dpcc** executable is self contained and doesn't reach for any implicit external asset and thus can be placed anywhere in the system and invoked from anywhere.

From now on we assume the user has a fired up shell correctly **cd**-ed to the directory holding the **dpcc** executable:

To call the compiler run the following command, which will print it's usage help message:

```
./dpcc
```

The **dpcc** compiler supports the specification of the `-o` flag where applicable. This flag allows to override the default output location.

dpcc can work in 6 different modes: **lex, parse, 3ac, c, gcc, run**:

- `./dpcc lex <input> [-o <out>]`: Lex the input and show the list of tokens composing the **DPL** source in either stdout or in the given file.
- `./dpcc parse <input> [-o <out>]`: Parse the program and produce a text representation of the AST (Abstract Syntax Tree) in either stdout or in the given file.
- `./dpcc 3ac <input> [-o <out>]`: Parse the program and perform additional type validations and type checking. If the program is still valid emit 3AC in either stdout or in the given file.
- `./dpcc c <input> [-o <out>]`: Same as 3AC but also emit preamble and postamble required to promote 3AC to a valid C program that can be compiled. The output is emitted in either stdout or in the given file.
- `./dpcc gcc <input> [-o <out>]`: Same as '`dpcc c`' but the generated C program is piped into GCC standard input and the final executable is either compiled in `a.out` or in the given filepath. This requires GCC to be in the path.
- `./dpcc run <input>`: Parse, typecheck, emit the C code, compile it and run it in one single command. The executable produced by GCC is outputted in a temp file (under `/tmp`), the temp executable is executed right away and then removed. The `-o` flag is ignored. This requires GCC to be in the path.

`Lex` and `parse` modes are mostly used for debugging and are not really that useful. The `run` mode is the most convenient mode since it takes care of everything. If the input program is valid and you call '`./dpcc run`' on it you will see the output generated from you **DPL** program, otherwise the compiler will complain with either warnings or errors.

---

[2] Github Repo Link

3 DPL Language Details

Providing a full language specification is beyond the scope of this project report. In particular this section will not list the entire grammar of the language. Thus, it is assumed that the reader has a common basic programming knowledge. It is also assumeed that he/she has some experience with at least one C-alike language. If the reader satisfies these requirements, he/she can use basic reasoning and code examples to deduce the specification of the language. Thus the purpose of this section is to characterize some core major concepts that distinguish **DPL** from other languages and that are not easily inferable:

- Comments start with ‘`//`’
- Identifiers start with a letter or an extened non ASCII UTF-8 character. After the first character an identifier can contain any alphanumerical character excluding spaces and punctuation characters. **Notice that names beginning with an underscore are reserved for compiler use and will be rejected**.
- Strings are enclosed in `"` (double quotes) and can contain valid ASCII escape sequences like in any traditional C-derived language.
- Print statement unlike in C are allowed to print any variable, and can deduce what should be printed based on the type of the variable that is passed.
- Most of the grammar is C-inspired, and in fact all control flow statements have the same syntax of C (or of any C-derived language).
- The precedence of the operators are taken directly from the C precedence table. The only **modification that DPL does differently than C** is that bitwise operators (`&, |, ^`) have higher precedence than the compare operators (`==, !=, ...`). Most modern languages adopt this convenient change, because it makes the precedence of the bitwise operators behave in the same way that normal mathematical operators work (`=, +, -, <<, >>`). In fact also **the Rust language employes** this same modification.
- A **DPL** program starts either in 2 ways. The first more idiomatic way is to just start writing statements directly:

  ```
  let a = 10;
  print(a + 20);
  ```

  The other way is to wrap **all** the statements inside a main function.

  ```
  fn main() {
      let = 10;
      print(a + 20);
  }
  ```

  Since DPL does not support functions yet the main function is mostly ignored but it is still part of the grammar for consistency reasons.
- **DPL** is a strongly typed language. Currently only 5 types are supported: `bool, int, float, string, bool[], int[], float[]`. As we talked about in previous sections `string` types behave a little bit different way.
- Code blocks are enclosed in braces ‘`{ ... }`’. Each code block define a new scope where variables can be defined.
- **Braces in control flow statements** (`if, while, for, ...`) **are always mandatory**. This is different from C where the braces are not mandatory. This change was done to simplify the grammar but also to avoid ambiguity and to make the code more robust to future changes. **Rust also imposes mandatory braces**.
- Variables can be declared with the keyword `let` and type deduction rules inside the compiler avoids the need to specify a type in most cases. A variable name must be a valid identifier. Variable declaration with the same variable name **can't** appear twice or more in the same code block. Reusage of variables names within nested blocks are instead allowed, even with different types: **variable shadowing**. Currently the reference compiler does not emit any warning in case a variable is shadowed.

4 DPCC compiler Implementation Details

This section describes how the `dpcc` compiler is composed: the input stage, the lexer stage, and the syntax analysis stage (parser), and the final code generator. **Flex** and **Bison** are used as tools for aiding in the boilerplate code generation of respectively the lexer and the parser.

The `dpcc` compilers models the entire program mostly using the following **C types**:

- `token_t` is basically a book-keeping type. It is meant to store metadata for each token. The most fundamental fields that it stores are: the `lexeme`, and the kind of each token (comment, identifier, string literal, ...). It also stores the location of each token within the file `(line:column)`
- `ast_node_t` is the **main core** type of the compiler. Multiple `ast_node_t`'s consitutes a full AST tree. When performing parsing using **Bison**, nodes are linked together in a parent/child relationship. Each node has the following fields:
  - A pointer to a token.
  - The node kind. The node kind is used to disambiguate the kind of the node (`Stmt, VarDeclStmt, Expr, ...`). It's one of the most important fields used in the code generation phase.
  - The codegen metadata. The codegen metadata is filled and used only in the last code generation phase of the compiler.
  - Multiple pointers to child nodes (if any).
  - A backpointer to the parent node (if any).
  - A pointer to the declaration node, used only for identifiers to lookup where they were declared.
  - A literal value. The literal value is used only in literals to store the value represented by this node (`int, float, bool`).
- `symtable_t` used to model the variables that are in scope. It's implementation for now is based on a linear array, and thus has linear search time performance. A hashmap could be used to improve the lookup performance.
- `ast_traversal_t` is a book-keeping context state used for traversing an AST.
- `codegen_ctx_t` is a context state for tracking already used 3AC variable names.

4.1 The input stage of the compiler

This is the simplest part of the compiler. At the time of writing the **dpcc** compiler allows only for the loading of files. In particular it reads an entire file into memory before continuining with the rest of the piepeline. The input stage does not support any form of **URI**, file downloads, any type of protocol that would require realtime on stream code generation, linux sockets, ...etc. That is the compiler can open anything that looks like a file that has a finite determinable start, an end and a finite number of bytes.

4.2 Lexer

Flex is used to implement the lexer/tokenizer. Lexers are pretty simple to understand and are particulary easy to develop if easing a tool like Flex. The unfamiliar reader with Flex can read the Flex Manual.

The things that are worth noting about the **dpcc** tokenizer are:

- The lexer is completely UTF-8 aware, and UTF-8 symbols can be used to declare identifiers. This is achieved by having particular Flex rules to match the variable encoding nature of UTF-8. **This allows a DPL program to declare variable names including emojis**. Why? Cause they're cool ☺
- The lexer tracks line and column locations thanks to the global variable `yylloc` exposed from Bison (when enabling `%locations`). This variable is updated accordingly in each rule, the line column and lines information are updated on each newline.
- Support for both Unix and MS-DOS style newlines.
- Support for C-style strings (can contain escape sequences).
- Support for C-style single line comments.

- Support for binary and hexadecimal integers. Support for C-style floating point numbers with the exponent and an optional terminating `'f'` character.

Flex is used mainly as a token recognizer since most of the logic is implemented outside the flex file anyway. One of the most notable feature that is implemented outside, is what's called **String interning**[3]. **String iterning** is a common technique used in compilers design that allows the compiler to store and cache lexemes in a common place. Since in typical source files the same lexemes tend to repeat and appear multiple times, it is common to store each lexeme only once. Whenever a lexeme is found it is looked up in a string to string hashmap. If it's not found, it allocates the new lexeme and returns the pointer to the new allocation. If it's found instead, it simply just returns the pointer to the interned lexeme. This allows to save memory, but even more cooler is the fact that now if two strings are identical and they are interned correctly, we can compare the two strings for equality by simply just comparing their respective pointers. In fact thanks to string interning strings are uniquely identified by the memory address they live in. This feature is then used in the parser to quickly lookup the identifiers in the symbol table.

For implementing **string interning** the amazing stb_ds.h single-file header from the awesome Sean Barrett's stb libraries is used as an hashmap implementation.

So upon encountering a new token the following things happen:

- Locations information (line, column) are updated correctly.
- The lexeme is interned, and the old lexeme pointer is stomped in favour of the interned one.
- A new `token_t` is allocated and filled with the lexeme pointer, the location and some other metadata.
- A new `ast_node_t` is allocated and the corresponding token is referenced. The `ast_node_t` will be used by Bison and later stages to generate the AST and to perform the analysis in the code generation phase. The node kind is instead left un-initialized for now. It will be set by the **Bison** parser later, where more semantic context is available.
- It signals to bison the new lexeme kind by simply `return`-ing from the `lex()` procedure (Bison calls flex in a coroutine mode)

4.3 Parser

Bison is used to aid in the implementation of the syntax analysis stage. The unfamiliar user can refer to the manual to learn about Bison. In the `dpcc` compiler **Bison** is used only for defining the grammar, and the `parse()` function generated from **Bison** is mostly used for syntax checking the input source.

In each **Bison** action the things worth noting that could occur are:

- A new `ast_node_t` is allocated and chained with other AST nodes in order to compose the full AST tree.
- Code blocks define a new scope where variable declarations can occur.
- A new variable declaration pushes a symbol into the symbol table.
- Each identifier used is looked up in the symbol table. If it's not found, then the parser errors out and refuses the program.

That is in `dpcc`, **Bison** is used only for defining the grammar and for checking if a program is syntactically valid. Apart for the extraordinary case that deals with the symbol table check & update, most of the compiler logic **is not implemented inside the bison file**.

Instead it is preferred to define most of the logic somewhere else, and to just setup the necessary AST to execute the logic later. This makes bison actions very simple and each action contain merely no code at all. In fact almost all actions in the bison file, call some C functions defined somewhere else: `NEW_NODE, push_child, push_childs`. These C functions are used to create a new node and to set it's type, and to append the corresponding childs to this new node. This keeps the bison file simple and concise since most of the heavyweight code is defined somewhere else.

The **advantage** of composing a full AST and deferring the execution of the logic to later stages are: simplicity, abstraction, code decoupling, and code scaling. The **disadvantage** instead, is that in order to emit the final code from the compiler, at least one more AST pass is required. Thus multiple passes could be potentially slower, than a single pass compiler. Also implementing difficult logic directly inside the bison file, or implementing all

---

[3] String Interning Wikipedia Article

the features required for a modern compiler (type deduction, type checking) in a single pass inside the **Bison file** becomes quickly tedious at best, if not nearly impossible.

This are the things worth noting about the **Bison** file used in the `dpcc` compiler:

- The `yylval` is aliased to be equivalent to `ast_node_t*`. Each node contains a field `kind` of type `ast_node_kind_t` which is initialized in the **Bison** file when a new node is created. This field allows later stages of the compiler to match what code should be generated for this node.
- Location tracking is turned (`%locations`). This allows **Bison** to declare the global variable `yylloc` which is used from the lexer to setup the location of each token.
- **LAC** (Lookahead Correction) mechanism is turned on. According to the Bison manual can provide better identification of the error location and thus better error messages, at the cost of a very negligable runtime speed penalty.
- Custom error reporting logging is enabled (`%define parse.error custom`). The function `yyreport_syntax_error` is defined accordingly. The implemented custom error reporting supports a GCC style error/warning and colored output and it is thus more user friendly. One noticeable feature is that **dpcc** compiler can warn when a variable is declared but never used.

4.4 Code Generation

This is the longest and most complicated part of the pipeline of the `dpcc` compiler. This section will be kept as concise as possible. Only the core concepts will be explained.

For simplicity reasons the current codegen implementation requires **two AST passes**. Whether the implementation of the code generation could be re-structured in order to do the same things in a **single pass** has not yet been tested and it is subject to discussion.

- **First AST pass**. In the first AST pass, three major things are done: type deduction, type checking and semantic validation. The input program may be refused due to either type mismatch or in general due to abuse of language features that cannot be easily checked in the parsing stage. As an example, it is not possible to declare a zero or negative sized array. Also, during this pass some metadata is initiliazed in each node. This metadata will guide the code generation in the second pass.
- **Second AST pass**. If the previous pass successed, we can assume that the code is syntatically valid and semantically valid. This is the pass where the actual 3AC gets outputted. This pass is **not allowed to fail**.

Type deduction works in the following easy to understand way:

1. **Base Cases**:
   (a) Statements, control flow and similar, do not have a meaningful type
   (b) The type of an `int`, `float`, `string`, ... **literal** are trivial.
   (c) The type that a casting operator produces is trivial. Casting operators force the conversion.
   (d) Variable delcarations **with a user listed type** are trivial. In this case type checking still needs to be performed in case the variable declaration has a `RHS` initialization. If the type check fails the compiler emits an error.
2. **Recursive Cases**:
   (a) Variable declaration with no user listed type inherit the type from the `RHS`. In case no `RHS` is provided, the compiler defaults the type to `int`.
   (b) The type of each expression is harder to deduce and check. Each expression has its own possible rules that describes which child types are allowed as inputs and which output type is generated for each case. If no rule match, than the compiler triggers a **type mismatch** error, otherwise it figures out the output type of the expression using a rule.

How all of this is achieved is thanks to two utility functions: `ast_traversal_begin, ast_traversal_next`. This two functions together allow to fully traverse each node composing the AST. It works in the following way: each time `ast_traversal_next` is called a pointer to a node and an integer index is returned with the following semantics:

1. **Base Case**. If the node is a leaf of the AST, `ast_traversal_next` returns the pointer to the node and an index set to `0 (zero)`.
2. **Recursive Case**. For each non-leaf node, `ast_traversal_next` will return the pointer to the node multiple times. The index is respectively set to:
    - `0 (zero)`. First time encountering this node. All childs still need to be visited.
    - `1`. Second time encountering this node. The first child was visited.
    - `2`. Third time encountering this node. The first, and second childs were visited.
    - `3`. Fourth time encountering this node. The first, second, and third childs were visited.
    - ...

    until all the number of childs are exhausted.

Code generation starts by operating on an empty string. Emission of new 3AC **is always concatenated** to the output string (like in a `printf`). The fact that new code must always be concatenated to the previous code is a very important concept to note and understand. It means that, when visiting each node of the AST, code must be generated and outputtted right away. This has the important consequence that **the order of the childs of each node is significant**. How the childs are ordered play an important role on the semantics and the validity of the final generated output. If one was willing to take a performance hit and implement in the compiler the ability to generate code on temporary strings, the compiler would then be able to combine those strings rearranging them in whichever order. At that point, the order of the childs wouldn't be relevant anymore.

The integer returned from `ast_traverse_next` is exploited in order to emit code snippets "in the middle" of partially evaluating/visiting a node. As an example of why this is important, the output that should be generated for a `while` statement wants to insert a **label** before evaluating the expression, and a **goto** to model the loop after the entire code block has been evaluated and emitted.

Here follows an example program and its associate compiled 3AC:

```
let a = [10, 20, 30, 40, 50];
let b = [1, 2, 3, 4, 5];
let result: int[5];

for (let i = 0; i < 5; i++) {
    result[i] = a[i] * b[i];
    if (i % 2 == 0) {
        print("YAAY -- ");
        print(i);
    }
}

print("Dot product result:\n");
print(result);
```

```
// Special variable used to implemenent INC (x++) and dec (x--)
// It is used to temporary hold the result of the INC/DEC in order to perform the side effect
int32_t _vspcIncDec;
// Special variable used for the negation of control statements (if, for, ...)
// As an example the for loop needs to negate the user provided condition
bool   _vspcNeg;

// 3AC Var decls
int32_t _vi0 = 0;
int32_t _vi1 = 0;
int32_t _vi2 = 0;
int32_t _vi3 = 0;
int32_t _vi4 = 0;
int32_t _vi5 = 0;
int32_t _vi6 = 0;
bool   _vb0 = false;
bool   _vb1 = false;

_scope_begin();
    _var_decl("a", _kI32, 5);
    _var_init("a", _kI32, 5, (int32_t[]) {10, 20, 30, 40, 50});
    _var_decl("b", _kI32, 5);
    _var_init("b", _kI32, 5, (int32_t[]) {1, 2, 3, 4, 5});
    _var_decl("result", _kI32, 5);
    _scope_begin();
```

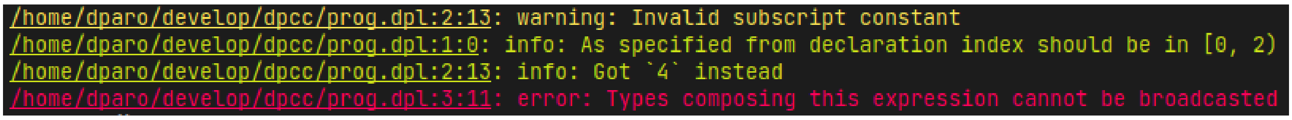
```
        _var_decl("i", _kI32, 1);
        _var_init("i", _kI32, 1, (int32_t[]) {0});
        _lbl2:
        _vb0 = _var_get_kI32("i", 0) < 5;
        _vspcNeg = !_vb0;
        if (_vspcNeg) goto _lbl3;
        _scope_begin();
            _vi0 = _var_get_kI32("result", _var_get_kI32("i", 0));
            _vi1 = _var_get_kI32("a", _var_get_kI32("i", 0));
            _vi2 = _var_get_kI32("b", _var_get_kI32("i", 0));
            _vi3 = _vi1 * _vi2;
            _vi4 = _var_set_kI32("result", _var_get_kI32("i", 0), _vi3);
            _vi5 = _var_get_kI32("i", 0) % 2;
            _vb1 = _vi5 == 0;
            _vspcNeg = !_vb1;
            if (_vspcNeg) goto _lbl1;
            _scope_begin();
                printf("%s", "YAAY -- ");
                print_sym("i");
            _scope_end();
            _lbl1:
        _scope_end();
        _vi6 = _var_get_kI32("i", 0);
        _vspcIncDec = _var_get_kI32("i", 0) + 1;
        _var_set_kI32("i", 0, _vspcIncDec);
        goto _lbl2;
        _lbl3:
    _scope_end();
    printf("%s", "Dot product result:\n");
    print_sym("result");
_scope_end();
```

4.5 Utilities and Misc

4.5.1 Custom logging

Custom logging is implemented to override the default Bison logging, and is used also manually in the type checking stage of the compiler to complain abot possible misuses. The output is inspireb from the **GCC** log output. Each message is colorized according to the severity, and it contains the filanem and the location of where the error/warning occurs. Here it follows an example program that makes the `dpcc` compiler complain:

```
let array: int[2];
print(array[4]);
print(3.5 << 8);
```



```
/home/dparo/develop/dpcc/prog.dpl:2:13: warning: Invalid subscript constant
/home/dparo/develop/dpcc/prog.dpl:1:0: info: As specified from declaration index should be in [0, 2)
/home/dparo/develop/dpcc/prog.dpl:2:13: info: Got `4` instead
/home/dparo/develop/dpcc/prog.dpl:3:11: error: Types composing this expression cannot be broadcasted
```

Figure 1: `dpcc` custom logging showoff

4.5.2 Typescript code generation

Without going too much deeper on the details, some part of the C source code composing the compiler is in fact generated using a **Typescript** program. Having now more knowledge about how the compiler source code turned out to be at the end, the decision of generating some part of the compiler turned out to be mostly **overkill**; especially considering the small size of the project. But anyway this trick is something that I wanted to try and see if it works. I didn't gain much using this code generation trick but I still think that theoretically speaking it could help in managing the source code of the compiler as it increases in scale.

Without spending too much time on this section, the **Typescript** program is used to do two things:

- To generate the code for the type deduction and type checking of the each expression operator. It takes some **meta-representations** of what each expression accept as input types and which output type it produces. Given these **meta-representations** it generates a C function called `typecheck_expr_and_operators` and some other utilities in a separate C file which is then compiled in the final executable. An example of such **meta-representation** is:

```
const MATH_EXPR = new Expr (
    [
        "ExprAdd", "ExprSub", "ExprMul", "ExprDiv", "ExprPow",
        "ExprInc", "ExprDec", "ExprPos", "ExprNeg",
    ],
    [
        new ExprTypeRule("int", ["int", "int"]),
        new ExprTypeRule("float", ["float", "int"]),
        new ExprTypeRule("float", ["int", "float"]),
        new ExprTypeRule("float", ["float", "float"]),

        new ExprTypeRule("int", ["int"]),
        new ExprTypeRule("float", ["float"]),
    ]
);
```

which rougly says that operators such as `+, -, *, /, **, ++, ...` can either take integers or floats as inputs, and depending on which input types are provided, it either produces an `int` or `float` type as output.

- To embed the required preamble and postamble C code inside the `dpcc` executable. The preamble and postamble code that are outputted when calling '`./dpcc c <input>`' are in fact written into two separate files. The typescript program reads these two files and generate two header files containing two `uint8_t[]` arrays that each encode the content of each respective file. Then these two generated `uint8_t` arrays are then embedded in the final executable.

4.5.3 Custom allocator wrapper

In order to track allocations inside the compiler a simple custom allocator is implemented. In practice this allocator just wraps the standard C allocator (`malloc`) and stores each allocation in a list. The reason for this is that one can simply allocate memory as he/she likes without worrying about freeing such memory. If the structure of the program is correctly thought out, one can simply define good synchronization points where it is safe to clear the entire allocator. Thus all allocations made up to this point can all be freed at once. One can also use multiple allocators to model different lifetime semantics for objects that must live longer or shorter.

The custom allocator lives in `src/utils.c` and the notable functions are:`dallnew, dallrsz, dalldel, dallclr, dallarr, ...`.

4.6 Testing framework

The `dpcc` compiler has unit testing framework setup to make sure that the compiler works as expected. The library Unity is a standalone unit framework written C. The `dpcc` uses this library to test some utilities freestanding functions in isolation.

Most of the testing horsepower is provided by a python script: `test/compile_test.py`. This script reads 2 files: `test/valid.dpl, test/invalid.dpl` which list respectively some valid and invalid dpl programs. Each program is separated by a long sequence of characters '`//`'. The python script extract each program separately, for each program it extract some metadata from the comments which list the expected output of the program. The python script then proceeds to call the compiler on that small program and verifies that either the program produces the expected output, or in the case of invalid programs it rejects it without crashing.

This is an example taken directly from `test/valid.dpl`:

```
//@ Boolean var decls
//@ t = 1
//@ 0

print("\n\nBoolean var decls\n");
{
    let t = true;
    print(t);
    print(false);
}

///////////////////////////////////////////////////////////////////////////////

//@ Integer array type deduction
//@ a = [ 10, 20, 30, 40, 50 ]
//@ a = [ 10, 20, 30, 40, 100 ]
```

```

print("\n\nInteger array type deduction\n");
{
    let a = [ 10, 20, 30, 40, 50 ];
    print(a);
    a[4] = 100;
    print(a);
}
////////////////////////////////////////////////////////////////////////////////
```

Notice the program separator and how the metadata is instead listed in a comment beginning with `//@`.

Here's instead some examples from `test/invalid.dpl`:

```
// Integer is too large
{
    let a = 10000000000000000000;
    print(a);
}
////////////////////////////////////////////////////////////////////////////////

// Arrays with no RHS must be sized
{
    let b: int[];
}
////////////////////////////////////////////////////////////////////////////////

// Arrays must have reasoanble size
{
    let a: int[-1];
}
////////////////////////////////////////////////////////////////////////////////

// Array with RHS must have correct size
{
    let a: int[3] = [ 2, 3 ];
}
////////////////////////////////////////////////////////////////////////////////
```

5 Performance results

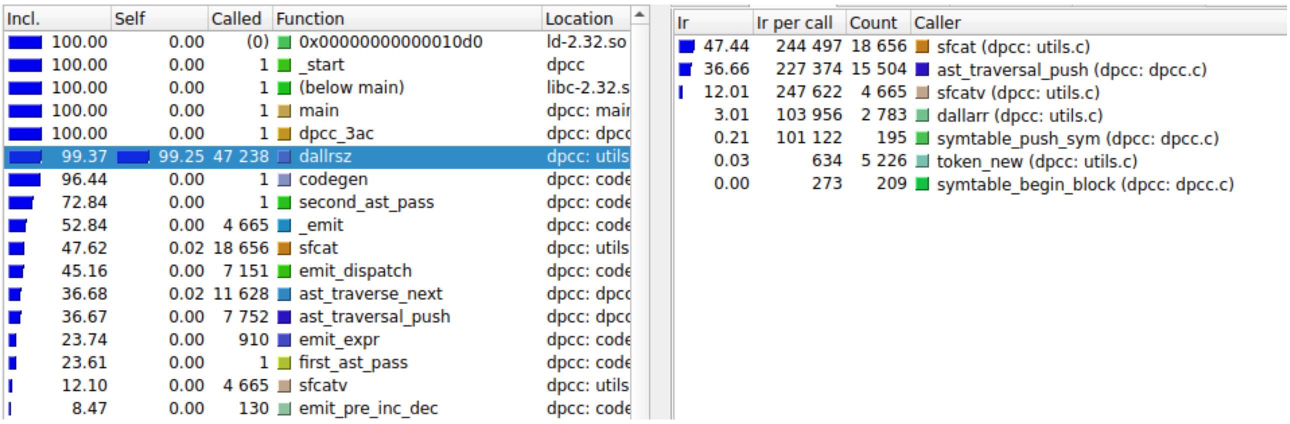
Valgrind is a very useful tool for C development. It is primarily used for memory debugging (memory corruptions, invalid writes acccess, . . .), finding memory leaks, and for profiling (timing, cache hit rate, branch prediction, . . .). Some memory bugs were totally eliminated inside the compiler thanks to this tool. Thanks to **valgrind**, also, some performance analysis abot the total running time of the compilar were analysed (`valgrind --tool=callgrind --dump-instr=yes --simulate-cache=yes --collect-jumps=yes -- ...`).

As it turned out from the analysis currently the compiler has un-satisfactory performance. The performance analysis highlighted the current custom allocator implementation is the **bottleneck** of the compiler. Most of the running time of the compiler is **wasted** on performing linear searches for allocation resizes.

As one can see from the below images, the `dallrsz` function, and in particular the linear scanning of the allocation consitutes more than **99%** of the total running time of the executable.

This problem is something that should totally be addressed before shipping the `dpcc` compiler to a final user.

Also I wanted to take some snapshots about the running time of the compilar as a function of the source code input size. Unfortunately since the current implementation of the allocator is **slow**, such snapshots wouldn't provide much information about how fast the compilation process is.



Incl.	Self	Called	Function	Location
100.00	0.00	(0)	0x00000000000010d0	ld-2.32.so
100.00	0.00	1	_start	dpcc
100.00	0.00	1	(below main)	libc-2.32.s
100.00	0.00	1	main	dpcc: mai
100.00	0.00	1	dpcc_3ac	dpcc: dpc
99.37	99.25	47 238	dallrsz	dpcc: utils
96.44	0.00	1	codegen	dpcc: code
72.84	0.00	1	second_ast_pass	dpcc: code
52.84	0.00	4 665	_emit	dpcc: code
47.62	0.02	18 656	sfcat	dpcc: utils
45.16	0.00	7 151	emit_dispatch	dpcc: code
36.68	0.02	11 628	ast_traverse_next	dpcc: dpc
36.67	0.00	7 752	ast_traversal_push	dpcc: dpc
23.74	0.00	910	emit_expr	dpcc: code
23.61	0.00	1	first_ast_pass	dpcc: code
12.10	0.00	4 665	sfcatv	dpcc: utils
8.47	0.00	130	emit_pre_inc_dec	dpcc: code

Ir	Ir per call	Count	Caller
47.44	244 497	18 656	sfcat (dpcc: utils.c)
36.66	227 374	15 504	ast_traversal_push (dpcc: dpcc.c)
12.01	247 622	4 665	sfcatv (dpcc: utils.c)
3.01	103 956	2 783	dallarr (dpcc: utils.c)
0.21	101 122	195	symtable_push_sym (dpcc: dpcc.c)
0.03	634	5 226	token_new (dpcc: utils.c)
0.00	273	209	symtable_begin_block (dpcc: dpcc.c)

Figure 2: Performance issue in `dallrsz` utility function.

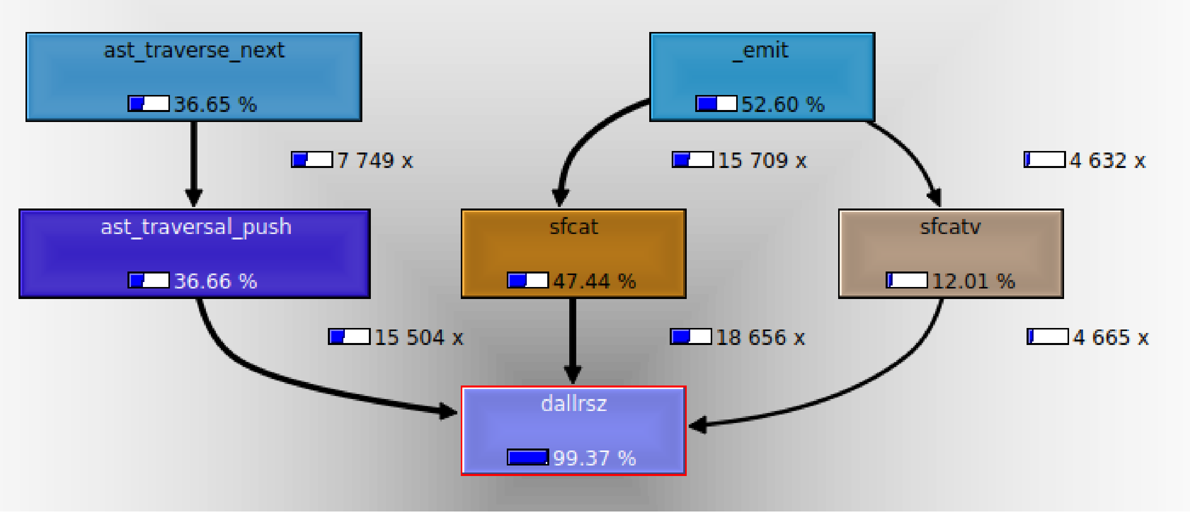
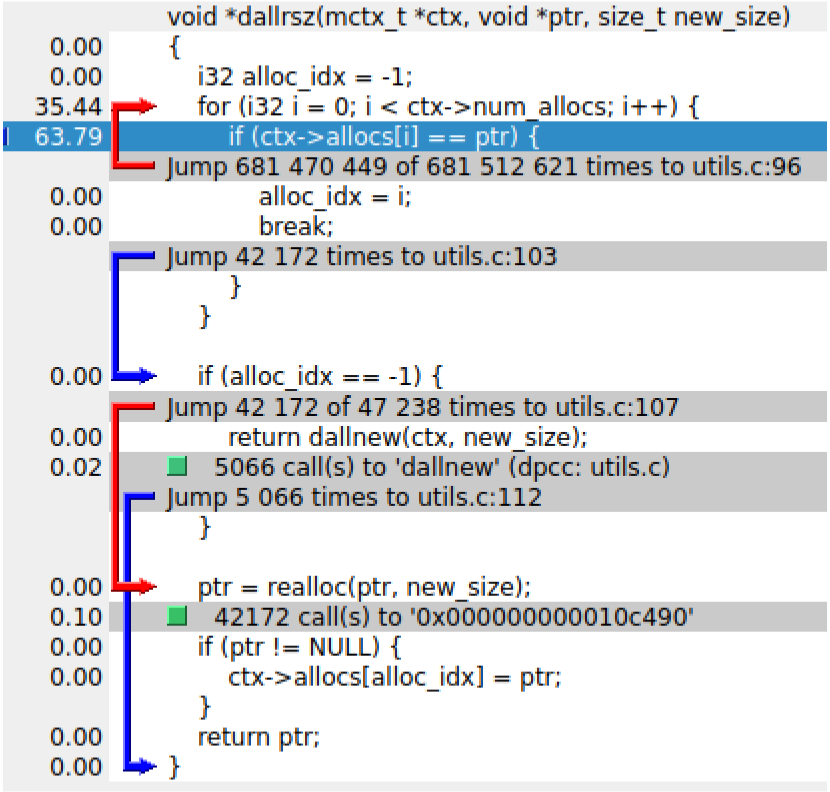


Figure 3: Performance issue same as previous image but in a graph form.



```
        void *dallrsz(mctx_t *ctx, void *ptr, size_t new_size)
 0.00   {
 0.00       i32 alloc_idx = -1;
35.44       for (i32 i = 0; i < ctx->num_allocs; i++) {
63.79           if (ctx->allocs[i] == ptr) {
        Jump 681 470 449 of 681 512 621 times to utils.c:96
 0.00               alloc_idx = i;
 0.00               break;
        Jump 42 172 times to utils.c:103
                }
            }

 0.00       if (alloc_idx == -1) {
        Jump 42 172 of 47 238 times to utils.c:107
 0.00           return dallnew(ctx, new_size);
 0.02   🟩  5066 call(s) to 'dallnew' (dpcc: utils.c)
        Jump 5 066 times to utils.c:112
            }

 0.00       ptr = realloc(ptr, new_size);
 0.10   🟩  42172 call(s) to '0x000000000010c490'
 0.00       if (ptr != NULL) {
 0.00           ctx->allocs[alloc_idx] = ptr;
            }
 0.00       return ptr;
 0.00   }
```

Figure 4: The linear search allocation scanning. The loop is taken tramendous amount of time for basically no useful reason.

6 Conclusions

The **DPL** language and the **dpcc** compiler are far from being useful and/or complete. They were implemented as part of a course in Compilers, so it is mostly a proof of concept. But still this proof of concept apply some modern features that languages like C lacks in its standard (type deduction, proper precedence table, proper fixed sized integers).

That being said, I still find that both this implementation and project report provide some concepts that are still applicable in a proper language & compiler implementation. The language mostly lacks proper features to be useful as an everyday productive language (procedure calls are a must). Also the compiler lacks a real-world efficient code generation backend. Unfortunately nowadays, due to the complexity of modern CPU architectures, writing a compiler backend is not an easy task. In fact most modern languages nowadays rely on external backends such as **LLVM**[4] to deal with the actual machine code output.

It would be cool to extend this language and bring it further. It would probably need some code refactoring/cleanup first, but the unit testing framework should help in that. Some cool concepts that could be investigated further are:

- **FUNCTIONS !!!**
- More basic types
- Custom definable types: struct, unions, possibly classes
- Namespaces to avoid the dependency hell that C has
- Proper metaprogramming system which is language and type aware (avoid C preprocessors macros)
- Proper module system
- Infrastructure: build system, package manager, tooling, and more ...
- ...

---

[4] LLVM Website

7 Appendix A: Example Program: Iterative Merge Sort

7.1 Input DPL source

```
let len = 32;
let array = [
    15, 59, 61, 75, 12, 71, 5, 35, 44,
    6, 98, 17, 81, 56, 53, 31, 20, 11,
    45, 80, 8, 34, 71, 83, 64, 28, 3,
    88, 50, 48, 80, 5
];


print("Un-sorted array\n");
print(array);

{
    for (let cs = 1; cs < len; cs = 2 * cs) {
        for (let l = 0; l < len - 1; l = l + 2 * cs) {
            let m = len - 1;
            if ((l + cs - 1) < len - 1) {
                m = l + cs - 1;
            }
            let r = len - 1;
            if ((l + 2 * cs - 1) < len - 1) {
                r = l + 2 * cs - 1;
            }


            let n1 = m - l + 1;
            let n2 = r - m;

            let L: int[1024];
            let R: int[1024];

            // Copy to temp arrays
            for (let i = 0; i < n1; i++) {
                L[i] = array[l + i];
            }
            for (let i = 0; i < n2; i++) {
                R[i] = array[m + 1 + i];
            }


            let i = 0;
            let j = 0;
            let k = l;
            while (i < n1 && j < n2) {
                if (!(L[i] > R[j])) {
                    array[k++] = L[i++];
                } else {
                    array[k++] = R[j++];
                }
            }

            while (i < n1) {
                array[k++] = L[i++];
            }
            while (j < n2) {
                array[k++] = R[j++];
            }
        }
    }
}

print("\nSorted array\n");
print(array);
```

7.2 Obtained output

```
Un-sorted array
array = [ 15, 59, 61, 75, 12, 71, 5, 35, 44, 6, 98, 17, 81, 56, 53, 31, 20, 11, 45, 80, 8, 34, 71, 83, 64, 28, 3,
      88, 50, 48, 80, 5 ]

Sorted array
array = [ 3, 5, 5, 6, 8, 11, 12, 15, 17, 20, 28, 31, 34, 35, 44, 45, 48, 50, 53, 56, 59, 61, 64, 71, 71, 75, 80,
     80, 81, 83, 88, 98 ]
```

7.3 Emitted 3AC code

```

// Special variable used to implemenent INC (x++) and dec (x--)
// It is used to temporary hold the result of the INC/DEC in order to perform the side effect
int32_t _vspcIncDec;
// Special variable used for the negation of control statements (if, for, ...)
// As an example the for loop needs to negate the user provided condition
bool   _vspcNeg;

// 3AC Var decls
int32_t _vi0 = 0;
int32_t _vi1 = 0;
int32_t _vi2 = 0;
int32_t _vi3 = 0;
int32_t _vi4 = 0;
int32_t _vi5 = 0;
int32_t _vi6 = 0;
int32_t _vi7 = 0;
int32_t _vi8 = 0;
int32_t _vi9 = 0;
int32_t _vi10 = 0;
int32_t _vi11 = 0;
int32_t _vi12 = 0;
int32_t _vi13 = 0;
int32_t _vi14 = 0;
int32_t _vi15 = 0;
int32_t _vi16 = 0;
int32_t _vi17 = 0;
int32_t _vi18 = 0;
int32_t _vi19 = 0;
int32_t _vi20 = 0;
int32_t _vi21 = 0;
int32_t _vi22 = 0;
int32_t _vi23 = 0;
int32_t _vi24 = 0;
int32_t _vi25 = 0;
int32_t _vi26 = 0;
int32_t _vi27 = 0;
int32_t _vi28 = 0;
int32_t _vi29 = 0;
int32_t _vi30 = 0;
int32_t _vi31 = 0;
int32_t _vi32 = 0;
int32_t _vi33 = 0;
int32_t _vi34 = 0;
int32_t _vi35 = 0;
int32_t _vi36 = 0;
int32_t _vi37 = 0;
int32_t _vi38 = 0;
int32_t _vi39 = 0;
int32_t _vi40 = 0;
int32_t _vi41 = 0;
int32_t _vi42 = 0;
int32_t _vi43 = 0;
int32_t _vi44 = 0;
int32_t _vi45 = 0;
int32_t _vi46 = 0;
int32_t _vi47 = 0;
int32_t _vi48 = 0;
int32_t _vi49 = 0;
int32_t _vi50 = 0;
int32_t _vi51 = 0;
int32_t _vi52 = 0;
int32_t _vi53 = 0;
```

```
int32_t _vi54 = 0;
int32_t _vi55 = 0;
int32_t _vi56 = 0;
int32_t _vi57 = 0;
bool   _vb0 = false;
bool   _vb1 = false;
bool   _vb2 = false;
bool   _vb3 = false;
bool   _vb4 = false;
bool   _vb5 = false;
bool   _vb6 = false;
bool   _vb7 = false;
bool   _vb8 = false;
bool   _vb9 = false;
bool   _vb10 = false;
bool   _vb11 = false;
bool   _vb12 = false;

_scope_begin();
    _var_decl("len", _kI32, 1);
    _var_init("len", _kI32, 1, (int32_t[]) {32});
    _var_decl("array", _kI32, 32);
    _var_init("array", _kI32, 32, (int32_t[]) {15, 59, 61, 75, 12, 71, 5, 35, 44, 6, 98, 17, 81, 56, 53, 31, 20,
        11, 45, 80, 8, 34, 71, 83, 64, 28, 3, 88, 50, 48, 80, 5});
    printf("%s", "Un-sorted array\n");
    print_sym("array");
    _scope_begin();
        _scope_begin();
            _var_decl("cs", _kI32, 1);
            _var_init("cs", _kI32, 1, (int32_t[]) {1});
            _lbl18:
            _vb0 = _var_get_kI32("cs", 0) < _var_get_kI32("len", 0);
            _vspcNeg = !_vb0;
            if (_vspcNeg) goto _lbl19;
            _scope_begin();
                _scope_begin();
                    _var_decl("l", _kI32, 1);
                    _var_init("l", _kI32, 1, (int32_t[]) {0});
                    _lbl16:
                    _vi0 = _var_get_kI32("len", 0) - 1;
                    _vb1 = _var_get_kI32("l", 0) < _vi0;
                    _vspcNeg = !_vb1;
                    if (_vspcNeg) goto _lbl17;
                    _scope_begin();
                        _var_decl("m", _kI32, 1);
                        _vi1 = _var_get_kI32("len", 0) - 1;
                        _var_init("m", _kI32, 1, (int32_t[]) {_vi1});
                        _vi2 = _var_get_kI32("l", 0) + _var_get_kI32("cs", 0);
                        _vi3 = _vi2 - 1;
                        _vi4 = _var_get_kI32("len", 0) - 1;
                        _vb2 = _vi3 < _vi4;
                        _vspcNeg = !_vb2;
                        if (_vspcNeg) goto _lbl1;
                        _scope_begin();
                            _vi5 = _var_get_kI32("l", 0) + _var_get_kI32("cs", 0);
                            _vi6 = _vi5 - 1;
                            _vi7 = _var_set_kI32("m", 0, _vi6);
                        _scope_end();
                        _lbl1:
                        _var_decl("r", _kI32, 1);
                        _vi8 = _var_get_kI32("len", 0) - 1;
                        _var_init("r", _kI32, 1, (int32_t[]) {_vi8});
                        _vi9 = 2 * _var_get_kI32("cs", 0);
                        _vi10 = _var_get_kI32("l", 0) + _vi9;
                        _vi11 = _vi10 - 1;
                        _vi12 = _var_get_kI32("len", 0) - 1;
                        _vb3 = _vi11 < _vi12;
                        _vspcNeg = !_vb3;
                        if (_vspcNeg) goto _lbl3;
                        _scope_begin();
                            _vi13 = 2 * _var_get_kI32("cs", 0);
                            _vi14 = _var_get_kI32("l", 0) + _vi13;
                            _vi15 = _vi14 - 1;
                            _vi16 = _var_set_kI32("r", 0, _vi15);
                        _scope_end();
                        _lbl3:
```

```
                        _var_decl("n1", _kI32, 1);
                        _vi17 = _var_get_kI32("m", 0) - _var_get_kI32("l", 0);
                        _vi18 = _vi17 + 1;
                        _var_init("n1", _kI32, 1, (int32_t[]) {_vi18});
                        _var_decl("n2", _kI32, 1);
                        _vi19 = _var_get_kI32("r", 0) - _var_get_kI32("m", 0);
                        _var_init("n2", _kI32, 1, (int32_t[]) {_vi19});
                        _var_decl("L", _kI32, 1024);
                        _var_decl("R", _kI32, 1024);
                        _scope_begin();
                            _var_decl("i", _kI32, 1);
                            _var_init("i", _kI32, 1, (int32_t[]) {0});
                            _lbl4:
                            _vb4 = _var_get_kI32("i", 0) < _var_get_kI32("n1", 0);
                            _vspcNeg = !_vb4;
                            if (_vspcNeg) goto _lbl5;
                            _scope_begin();
                                _vi20 = _var_get_kI32("L", _var_get_kI32("i", 0));
                                _vi21 = _var_get_kI32("l", 0) + _var_get_kI32("i", 0);
                                _vi22 = _var_get_kI32("array", _vi21);
                                _vi23 = _var_set_kI32("L", _var_get_kI32("i", 0), _vi22);
                            _scope_end();
                            _vi24 = _var_get_kI32("i", 0);
                            _vspcIncDec = _var_get_kI32("i", 0) + 1;
                            _var_set_kI32("i", 0, _vspcIncDec);
                            goto _lbl4;
                            _lbl5:
                        _scope_end();
                        _scope_begin();
                            _var_decl("i", _kI32, 1);
                            _var_init("i", _kI32, 1, (int32_t[]) {0});
                            _lbl6:
                            _vb5 = _var_get_kI32("i", 0) < _var_get_kI32("n2", 0);
                            _vspcNeg = !_vb5;
                            if (_vspcNeg) goto _lbl7;
                            _scope_begin();
                                _vi25 = _var_get_kI32("R", _var_get_kI32("i", 0));
                                _vi26 = _var_get_kI32("m", 0) + 1;
                                _vi27 = _vi26 + _var_get_kI32("i", 0);
                                _vi28 = _var_get_kI32("array", _vi27);
                                _vi29 = _var_set_kI32("R", _var_get_kI32("i", 0), _vi28);
                            _scope_end();
                            _vi30 = _var_get_kI32("i", 0);
                            _vspcIncDec = _var_get_kI32("i", 0) + 1;
                            _var_set_kI32("i", 0, _vspcIncDec);
                            goto _lbl6;
                            _lbl7:
                        _scope_end();
                        _var_decl("i", _kI32, 1);
                        _var_init("i", _kI32, 1, (int32_t[]) {0});
                        _var_decl("j", _kI32, 1);
                        _var_init("j", _kI32, 1, (int32_t[]) {0});
                        _var_decl("k", _kI32, 1);
                        _var_init("k", _kI32, 1, (int32_t[]) {_var_get_kI32("l", 0)});
                        _lbl10:
                        _vb6 = _var_get_kI32("i", 0) < _var_get_kI32("n1", 0);
                        _vb7 = _var_get_kI32("j", 0) < _var_get_kI32("n2", 0);
                        _vb8 = _vb6 && _vb7;
                        _vspcNeg = !_vb8;
                        if (_vspcNeg) goto _lbl11;
                        _scope_begin();
                            _vi31 = _var_get_kI32("L", _var_get_kI32("i", 0));
                            _vi32 = _var_get_kI32("R", _var_get_kI32("j", 0));
                            _vb9 = _vi31 > _vi32;
                            _vb10 = ! _vb9;
                            _vspcNeg = !_vb10;
                            if (_vspcNeg) goto _lbl8;
                            _scope_begin();
                                _vi33 = _var_get_kI32("k", 0);
                                _vspcIncDec = _var_get_kI32("k", 0) + 1;
                                _var_set_kI32("k", 0, _vspcIncDec);
                                _vi34 = _var_get_kI32("array", _vi33);
                                _vi35 = _var_get_kI32("i", 0);
                                _vspcIncDec = _var_get_kI32("i", 0) + 1;
                                _var_set_kI32("i", 0, _vspcIncDec);
                                _vi36 = _var_get_kI32("L", _vi35);
```

```
                                _vi37 = _var_set_kI32("array", _vi33, _vi36);
                            _scope_end();
                            goto _lbl9;
                            _lbl8:
                            _scope_begin();
                                _vi38 = _var_get_kI32("k", 0);
                                _vspcIncDec = _var_get_kI32("k", 0) + 1;
                                _var_set_kI32("k", 0, _vspcIncDec);
                                _vi39 = _var_get_kI32("array", _vi38);
                                _vi40 = _var_get_kI32("j", 0);
                                _vspcIncDec = _var_get_kI32("j", 0) + 1;
                                _var_set_kI32("j", 0, _vspcIncDec);
                                _vi41 = _var_get_kI32("R", _vi40);
                                _vi42 = _var_set_kI32("array", _vi38, _vi41);
                            _scope_end();
                            _lbl9:
                        _scope_end();
                        goto _lbl10;
                        _lbl11:
                        _lbl12:
                        _vb11 = _var_get_kI32("i", 0) < _var_get_kI32("n1", 0);
                        _vspcNeg = !_vb11;
                        if (_vspcNeg) goto _lbl13;
                        _scope_begin();
                            _vi43 = _var_get_kI32("k", 0);
                            _vspcIncDec = _var_get_kI32("k", 0) + 1;
                            _var_set_kI32("k", 0, _vspcIncDec);
                            _vi44 = _var_get_kI32("array", _vi43);
                            _vi45 = _var_get_kI32("i", 0);
                            _vspcIncDec = _var_get_kI32("i", 0) + 1;
                            _var_set_kI32("i", 0, _vspcIncDec);
                            _vi46 = _var_get_kI32("L", _vi45);
                            _vi47 = _var_set_kI32("array", _vi43, _vi46);
                        _scope_end();
                        goto _lbl12;
                        _lbl13:
                        _lbl14:
                        _vb12 = _var_get_kI32("j", 0) < _var_get_kI32("n2", 0);
                        _vspcNeg = !_vb12;
                        if (_vspcNeg) goto _lbl15;
                        _scope_begin();
                            _vi48 = _var_get_kI32("k", 0);
                            _vspcIncDec = _var_get_kI32("k", 0) + 1;
                            _var_set_kI32("k", 0, _vspcIncDec);
                            _vi49 = _var_get_kI32("array", _vi48);
                            _vi50 = _var_get_kI32("j", 0);
                            _vspcIncDec = _var_get_kI32("j", 0) + 1;
                            _var_set_kI32("j", 0, _vspcIncDec);
                            _vi51 = _var_get_kI32("R", _vi50);
                            _vi52 = _var_set_kI32("array", _vi48, _vi51);
                        _scope_end();
                        goto _lbl14;
                        _lbl15:
                    _scope_end();
                    _vi53 = 2 * _var_get_kI32("cs", 0);
                    _vi54 = _var_get_kI32("l", 0) + _vi53;
                    _vi55 = _var_set_kI32("l", 0, _vi54);
                    goto _lbl16;
                    _lbl17:
                _scope_end();
            _scope_end();
            _vi56 = 2 * _var_get_kI32("cs", 0);
            _vi57 = _var_set_kI32("cs", 0, _vi56);
            goto _lbl18;
            _lbl19:
        _scope_end();
    _scope_end();
    printf("%s", "\nSorted array\n");
    print_sym("array");
_scope_end();
```