Waseda University

Computer Language Processors' Compiler "tlc" Functional Specification

Keiji Kimura

March, 2016 (Japanese Ver.)

June, 2020 (English Ver.)

1. Introduction

This document describes the functional specification for the Computer Language Processors' compiler "tlc". tlc consists of a lexical analyzer, a parser, and a code generator for x86-64 and ARM64. On the other hand, it does not include optimizers and well-constructed error handlers to simplify the implementation. The compiler can be executed on Raspberry Pi4 (ARM64), Linux and Mac, and it can generate the 64-bit code for each platform.

2. Building Compiler

Firstly, extract the distributed "tlc.zip":

```
unzip tlc64.zip
```

Then, you can find the directory "tlc64" and extracted directories and files in it. It contains the following directories:

- tlc64/
- tlc64/doc/
  - Directory for document files
- tlc64/src/
  - Directory for source files
- tlc64/src/test
  - Directory for test files written in the language "tl", which is the source language of tlc.

To build tlc, you need to modify the lines of "PLATFORM=" in Makefile in tlc64/src directory. For instance, if you use Linux (including WSL), remove "#" at the beginning of the "PLATFORM = LINUX", and put "#" at the beginning of other "PLATFORM=" lines. For Raspberry Pi4, remove "#" at the line of "RASPI". For Apple silicon (i.e., M1) Mac, remove "#" at the line of "ARMMAC". Then, execute "make" command, and you can get the executable binary file of tlc.

```
make
```

(You need to execute make again to build the updated tlc when you modify the source code.)

To check tlc, you can compile the prepared test programs:

```
cd test
../tlc test1.c
```

You can get test1.s. It can be compiled by gcc.

```
gcc –o test1 test1.s
./test1
```

3. Executing compiler

You can execute tlc with the file name of a source program. The suffix of a source program is ".c" because the language specification of tl is a subset of C language. For instance, you can compile a source program "sample.c" as the following:

```
tlc sample.c
```

In the generated file as a result of the compilation, the suffix is substituted from ".c" to ".s", which represents the

generated file is assembly of x86-64 or ARM64. For instance, the compiler generates sample.s from sample.c. To obtain an executable binary, you can use gcc:

```
gcc sample.s
```

The example above generates the binary file named “a.out”. To specify the name of the executable binary file, you can use “-o” option. To specify the name “sample”, “gcc -o sample sample.s” can be used.

4. Organization of Compiler

4.1. Lexical Analyzer

It performs lexical analysis. It recognizes the following tokens:

- **Punctuators**
  - **+,-,*,/,<,<=,>=,>,==,!=, , (comma) , ((open parenthesis) ,) (close parenthesis) , ;**
- **Integer constants**
  - **A series of “0-9”**
- **Identifiers**
  - **Beginning from “a-z” or “A-Z”, and a series of “a-z”, “A-Z”, “0-9”, or “_” more than 0.**
- **Keywords**
  - **else, for, if, int, main, return, while**

tlc uses flex to generate lexical analyzer.

4.2. Parser

It performs syntax analysis along with the language specification, then generates an abstract syntax tree (AST) as the result. tlc uses bison to generate the parser.

4.3. Code Generator

It generates x86-64 or ARM64 assembly from AST.

5. Intermediate Representation

The intermediate representation of tlc is an Abstract Syntax Tree (AST). Nodes in the AST are categorized into the following groups:

- Function
  - A function node has a doubly linked list of statements as the body of the function. Functions in a compilation unit is maintained in a doubly linked list. Each function has its id number.
- Statement
  - A statement is an assign statement, if-statement, while-statement, for-statement, or return statement. A statement has expressions or a list of statements as its children.
- Expression

- Expression is one of an identifier, an integer constant, an expression surrounded by parentheses, an unary expression, a multiply/divide expression, an add/sub expression, a comparison expression, an equality expression, or assign expression. An expression has operands as its children.

6. Internal Data Structure

6.1. Symbol Table

The symbol table maintains variables. It contains local variables for each function with their offset in a stack frame.

6.2. Abstract Syntax Tree (AST)

An AST consists of nodes each of which represents a function, a statement, or an expression. Each node can access a parent node.

AST_kind represents a kind of a node in an AST. It can be one of the following kinds:

- AST_KIND_FUNC
  - Function
- AST_KIND_STM
  - Statement
- AST_KIND_EXP
  - Expression

A node can have at most four child nodes and a list of statements.

A node also has a sub-kind, AST_sub_kind, as the following:

- For AST_KIND_STM
  - AST_STM_LIST
    - List of statements
  - AST_STM_ASIGN
    - The first child is an assign expression
  - AST_STM_IF
    - The first child is a condition expression, the second child is a list of statements for “then”part, and the third child is a list of statements for “else” part.
  - AST_STM_WHILE
    - The first child is a controlling expression, which controls whether loop continues, and the second child is a list of statements in the loop body.
  - AST_STM_FOR
    - The first child is an initialize expression, the second child is a controlling expression, the third child is an expression executed at the end of each iteration, and the fourth child is a list of statements in the loop body.

- AST_STM_DOWHILE
  - The first child is a list of statements in the loop body and the second child is a controlling expression.
- AST_STM_RETURN
  - The first child is a return expression.

– For AST_KIND_EXP
- AST_EXP_ASGN
  - The first child is the left-hand side expression, and the second child is the right-hand side expression.
- AST_EXP_IDENT
  - A node contains a value corresponding with a variable name in the symbol table.
- AST_EXP_CNST
  - A node contains a constant value.
- AST_EXP_PRIME
  - The first child is an expression surrounded by parentheses.
- AST_EXP_CALL
  - A function-call. The first child is a node of AST_EXP_IDENT representing the function name. Parameters are kept in a list.
- AST_EXP_UNARY_PLUS
  - The first child is an expression.
- AST_EXP_UNARY_MINUS
  - The first child is an expression.
- AST_EXP_MUL
  - The first and the second children are operand expressions. The following expression nodes also have two operands expressions as children.
- AST_EXP_DIV
- AST_EXP_ADD
- AST_EXP_SUB
- AST_EXP_LT
- AST_EXP_GT
- AST_EXP_LTE
- AST_EXP_GTE
- AST_EXP_EQ
- AST_EXP_NE

Statements in a function body and a loop body for a loop, such as while-loop, are kept in a doubly linked list. Functions in a compilation unit are also kept in a doubly linked list.

7. Test of tlc

tlc/src/test contains test files for tlc. dotest.sh in this directory is prepared to test the compiler for each platform. For Linux (including WSL), this script can be used as the following:

```
./dotest.sh LIN
```

This generates compile logs, assembly files, and executable binary files in "tmp" directory. The log files and the assembly files are compared with the reference files prepared in "tlc/src/test/LIN" directory. The script reports differences if it finds, otherwise the script gives no reports. For Mac, "MAC" can be used instead of "LIN". For Raspberry Pi4, "RPI" can be used. For Apple silicon Mac, "AMAC" can be used.

8. Source files for tlc

- **tl_lex.l**
  - The definition of the lexical analysis
- **tl_gram.y**
  - The definition of syntax analysis
- **util.[ch]**
  - Utility function such as memory allocation
- **symtab.[ch]**
  - Symbol table management
- **ast.[ch]**
  - AST management
- **parse_action.[ch]**
  - Action functions for the parser
- **cg.[ch]**
  - Code generation (for target independent part)
- **arch_common.h, arch_x64.c, arch_arm64.c**
  - Target dependent part (for memory assignment and code generation)
- **main.c**
  - Main routine