CS 5363: Compiler Project – Spring - 2015
Software Design Document

Compiler for TL language

Stakeholders:

Prof. Xiaoyin Wang

Developer:

Rajiv Shanmugam Madeswaran – cxk437

Document Modification History

Version	Date	Author	Description
1.0	5/03/2016	Rajiv Shanmugam Madeswaran	Final Version
1.1			
1.2			
1.3			
1.4			
1.5			
1.6			

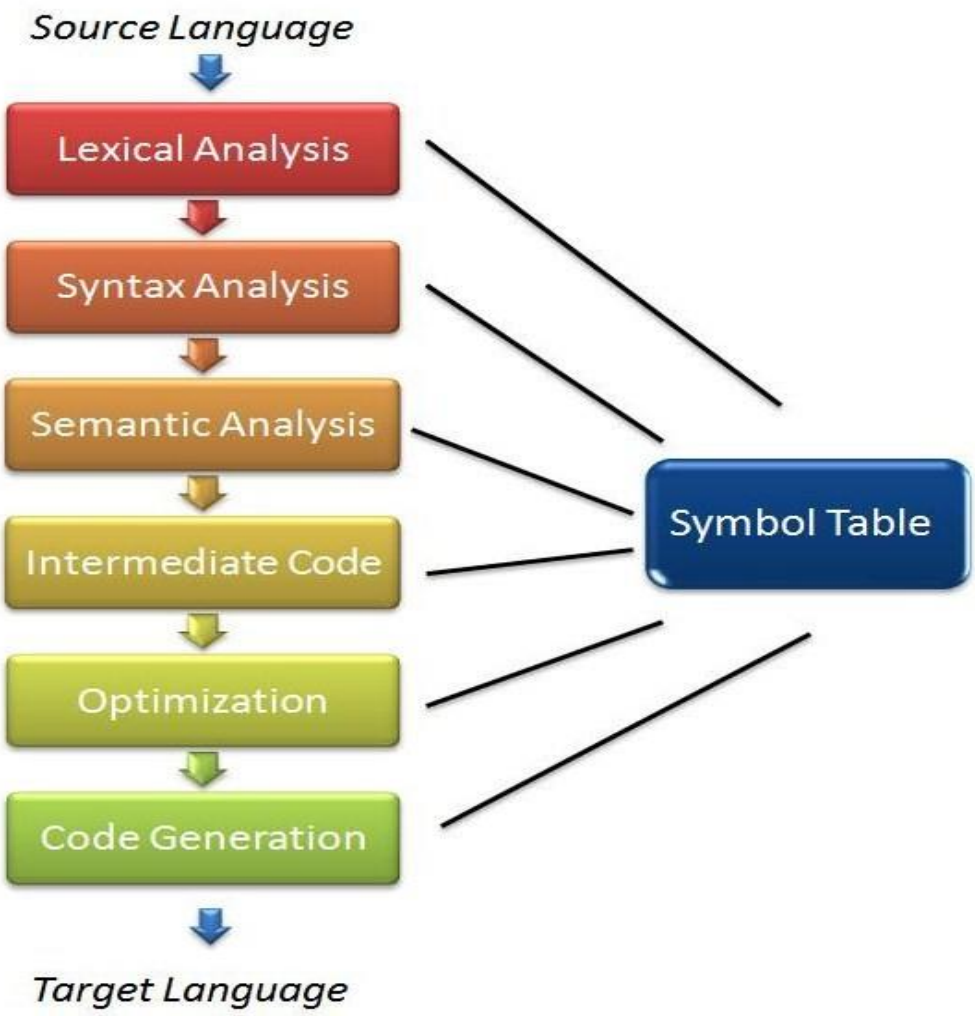
1. Purpose

The software design document describes the design of the TL compiler developed using C programming Language.

2. Scope

By using the TL compiler anyone can write programs as per the TL language specification and then compile it to generate the assembly code. Then this assembly code can be evaluated using QtSpim MIPS simulator.

3. Compiler Architecture



Lexical Analysis:

This is the first phase of the scanner. It scans the source code in to a stream of characters and converts them in to an atomic unit called lexemes or tokens.

Syntax Analysis:

This phase is known as parsing. This takes token as an input from the lexical analysis and generates parse or syntax tree. In this phase token arrangements are checked with BNF grammar of the Toy Language specification.

Semantic Analysis:

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation:

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language.

Optimization:

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources.

Code Generation:

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language.

Symbol Table:

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it.

4. Component Design Structures:

Lexical Analysis Component Structures:

C Files related to Scanner: scanner.c, double_link.c
Header Files related to scanner part: double_link.h, scanner_codes.h, scanner.h

Important Structure:

```
struct token_info {
        unsigned char text[SIZE]; /* stores the atomic token by scanning the source code */
        unsigned pos; /* Position of the token */
        unsigned line_no; /* line number of the token */
        unsigned token_type; /* type to identify its class */
        struct token_info *next; /* doubly linked list of tokens form the source code */
        struct token_info *prev;
};
```

The above structure will stores the sequence of tokens which are collected from scanning the source code. This token_info doubly linked list is passed on to the parser to check whether the token layouts are as per the Toy Language specification.

Output of scanner Phase:
Collection of tokens retrieved and stored it in a <basename.tok> file.

Parser Component Structures:

C Files related to Parser: parser.c, tree_op.c, draw_tree.c
Header File related to Parser: parser.h, tree_ops.h

Important Structure:

```
struct tree_node {
        struct tree_node *left, *right; /* pointers to left and right child of the tree */
        struct token_info tok; /* token residing in the particular tree node */
        int node_id; /* node id for DOT file generation */
        int type; /* Token type of the node used if it is an operator */
        int offset; /*Used to note the frame offset for ASM generation */
        int visited; /*flag for visitng the node only once */
        int reg; /* temporary register where the value is stored */
        int error; /* flag if there is any type checking Error */
};
```

The parser phase will construct parser tree which are in conformance with the BNF grammar. This structure forms a tree for the source code given at run time. The above tree is annotated with the type checking rules as mention in the TL language specification.

Output of the parser phase:
This phase generates DOT file which can be used to visualize the tree generated by the parser. This Compiler will generate AST.dot file. The visualized tree also depicts if there is any violation in type rules.

Code Generation Component Structures:

C Filed related to code generation: asm_gen.c, create_cfg.c

This phase directly generates an assembly code for the TL programming language. It is done by traversing the AST by in-order. After the assembly code generation, a DOT file for the control flow graph is created. This helps the user to visualize the blocks and its control flow for the given input TL source code.

Output of the code generation phase:

This phase generates an assembly code in the name of Assembly_Code.s. From this assembly code the control flow graph is created to visualize the control flow

of the user program. The Control flow graph is created from a DOT file naming Control_Flow.dot.

Symbol Table Component Structure:

Important Structure:

```
struct sym_entry {
        char token[SIZE]; /* Token text stored in the entry */
        unsigned type; /* Type of the token i.e keyword or number etc */
        unsigned reg; /* register its value is presently stroed */
        struct sym_entry *next; /* In case of collision, list of entries is created */
        int frame_offset; /* Stack frame information */
        int valid; /* valid entry or not */
};
```

The symbol table for the TL compiler is implemented by using the Hash Table. The Identifiers, Keywords are indexed in to the hash table as per the starting character of these words. In case of collision on any of the buckets the corresponding entry is chained in to the particular bucket forming the list of entries. So, the number of buckets in the hash table corresponds to the number of alphabets.

5. Steps to Use the Compiler

The TL compiler first need to be compiled in order to use it with the TL program.

- Extract the given compiler project. After extracting, two directories will be created in the current directory called “src” and “workdir”
- In the “src” directory all the code related to the compiler can be accessed
- In the “workdir” there will be two shell scripts called “build.sh” and “execute.sh”
- “build.sh” is used to compile the code which are in the source directory.
- “execute.sh” is will run the compiler along with the source program given as its input

6. Screenshots for each phase of the Compiler

Screenshots below are for the input TL program which computes the sum of 'N' numbers.

Compilation Step:

```
[root@localhost workdir]# sh build.sh

tl_compiler ELF created..

Please run exec.sh
```

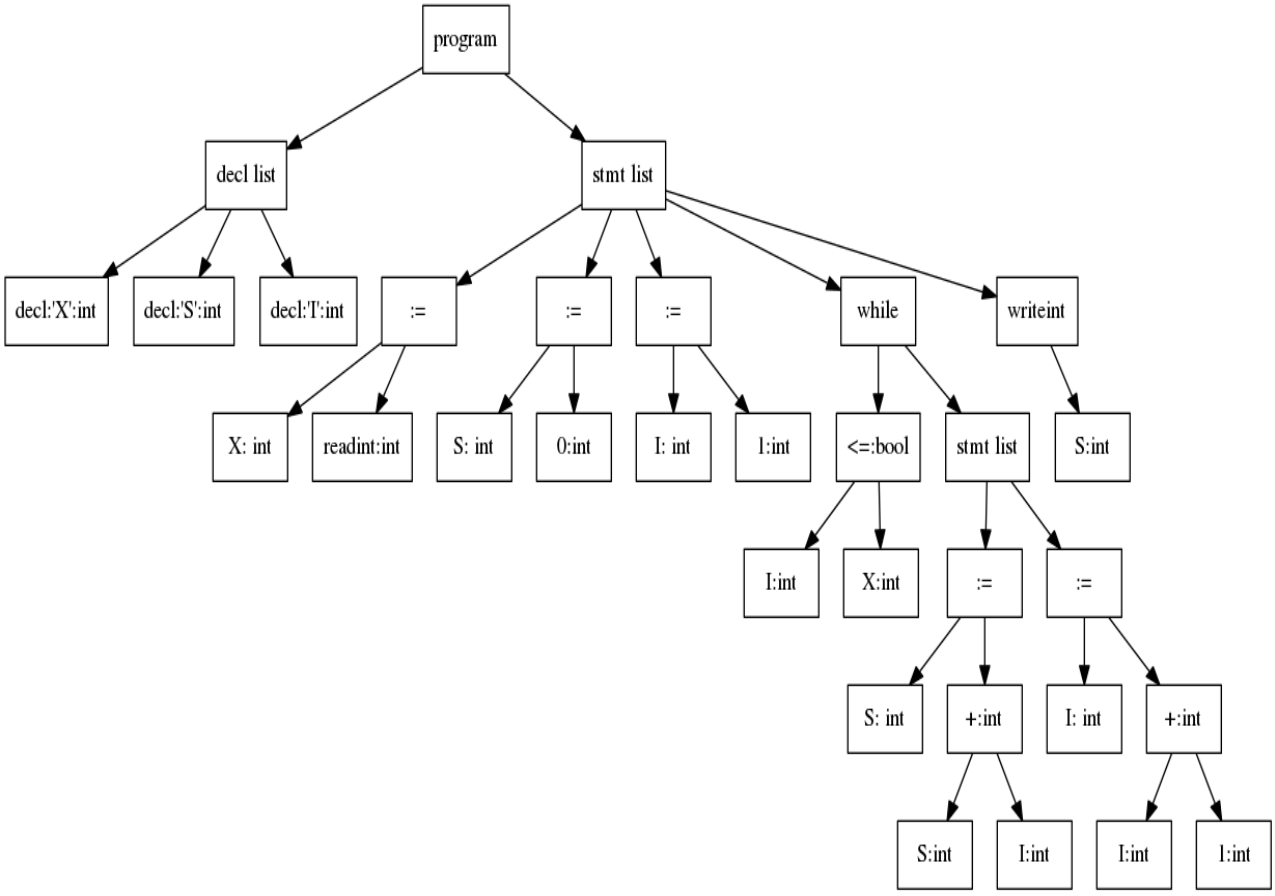
Execution step:

```
[root@localhost workdir]# sh exec.sh sum_n.tl

Completed: created scanner output in sum_n.tok
Completed: created AST.dot file in the current directory
Completed: created Control_Flow.dot file in the current directory
Completed: created Assembly_Code.s file in the current directory
sum_n.tl.ast.png file created for AST.dot file
sum_n.tl.asm.cfg.png file created for Control_Flow.dot file

[root@localhost workdir]# ls
Assembly_Code.s  build.sh          exec.sh  Makefile      prime.tl  sum_n.tl.asm.cfg.png  sum_n.tok
AST.dot          Control_Flow.dot  fib.tl   oddOReven.tl  sum_n.tl  sum_n.tl.ast.png      tl_compiler
[root@localhost workdir]#
```

Annotated Abstract Syntax Tree from DOT file:



Control Flow Graph from DOT file: