ECE 449
Project 2: Syntactic Analysis
Linghan Zhao A20313766
Oct 9, 2014

Syntactic Analysis

1. Overview

In project 2, syntactic analysis has been designed and implemented based on the previous project lexical analysis. In the first project, we extract tokens from EasyVL file and store them into .token files. In this project, I perform syntactic analysis, i.e. to study the grammar of the EasyVL language that in particular defines wires, components, and pins. The main function is to move tokens to statements and use finite statement to extract wire and component. Finally, display the corresponding syntactic file contains all classified types and contents.

This project is developed in eclipse C++ environment and executed both in windows and CI system.

2. Design

Initial Release Features

In the initial release, I refactor code, using functions and data structure, based on the previous project. Break a long program into smaller pieces so each piece is responsible for a straightforward matter. I use functions: extract tokens from a line, extract tokens from a file, display tokens on screen, store tokens to a file and so on. The main functions show blow.

Name	Type	Functions
extract_tokens_from_line	bool	Extract tokens from lines and classify them by NAME, NUMBER and SINGLE. (project 1 main function)
extract_tokens_from_file	bool	Read file and extract tokens using extract_tokens_from_file function
display_tokens	void	display tokens with token type
store_tokens_to_file	bool	store tokens using display_tokens and store them to .token file

To move tokens to statements and group them to display, in the initial release, I choose to use vector data structure to store tokens and statements. Then finish the ENDMODULE and MODULE statement and display them successfully. In this part, identifying modules, endmodule, wires, and components from the tokens is the most important. The main structure and functions show below.

Name	Type	Functions
struct evl_token (typedef std::vector<evl_token> evl_tokens;)	vector	vector data structure to store tokens
struct evl_statement (typedef std::vector<evl_token> evl_tokens;)	vector	vector data structure to store statements
move_tokens_to_statement	bool	move tokens to statement by lines using ; to seperate
group_tokens_to_statment	bool	Identify tokens by group and put them to corresponding vectors. Implement MODULE and ENDMODULE part
display_statements	void	Display statements based on required format.
store_statements_to_file	bool	store statements using display_statements and store them to .syntax file

The reason why I choose this features because refactoring codes of first project will be helpful for further functions implementation. Also it makes codes more readable and extendable. And the MODULE and ENDMODULE are the basic parts to implement.

Final Release Features

In the final version, I finish all the rest part including wire, component and display format part. At first, I introduce vector data structure to wire, component and component pins. Then, I define two functions to identify wire and component in finite state machine way. And store them to corresponding vectors. One more thing, for the time complexity, I change the data structure of token and statement from vector to list. Finally, use corresponding access ways to display them in the right format. The main functions and structure show below.

Name	Type	Functions
struct evl_token (typedef std::list<evl_token> evl_tokens;)	list	Change vector to list data structure to store tokens
struct evl_statement (typedef std::list<evl_token> evl_tokens;)	list	Change vector to list data structure to store statements
struct evl_wire (typedef std::vector<evl_wire> evl_wires;)	vector	vector data structure to store wires
struct evl_component (typedef std::vector<evl_component> evl_component;)	vector	vector data structure to store components
struct evl_pin (typedef std::vector<evl_pin> evl_pins;)	vector	vector data structure to store pins

process_wire_statement	bool	Using finite-state machine to extract wire tokens
process_component_statement	bool	Using finite-state machine to extract component, pins, and store them to corresponding structure
group_tokens_into_statements	bool	Finish rest part of wire and component, and former two functions to group wire, component and store them to statement.
display_statements	void	Finish rest part of wire and component branches, access them by corresponding structure and display them in correct format

3. Key Implement

This project focuses on the syntactic analysis of EasyVL language. And what I should do based on previous project is to convert the tokens to four type (MODULE, wire, component and ENDMODULE) statements and store them in specific format in .syntax file. MODULE and ENDMODULE part is not hard to implement as showed before in first release features. The key implement part of this project is to design wire and component statement part. Both of them will be implemented by a finite-state machine (FSM) loop.

The FSM for wires is shown if Figure 1. At every loop, the FSM will first be initialed to INIT. And then the FSM will be determined by specific condition. The whole process is implemented in function process_wire_statement.

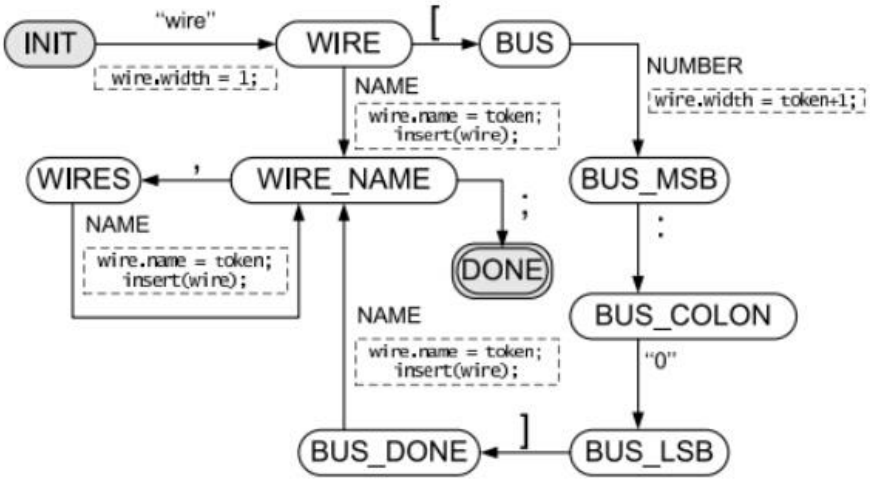

Figure 1. FSM for wires

The FSM for components is shown if Figure 2. The FSM is much same as wires', but the components have pins part to handle and process. The pins and components will be identified and stored. As the FSM flow, we store pins to pin

vector of component vector, which are both defined in data structure.

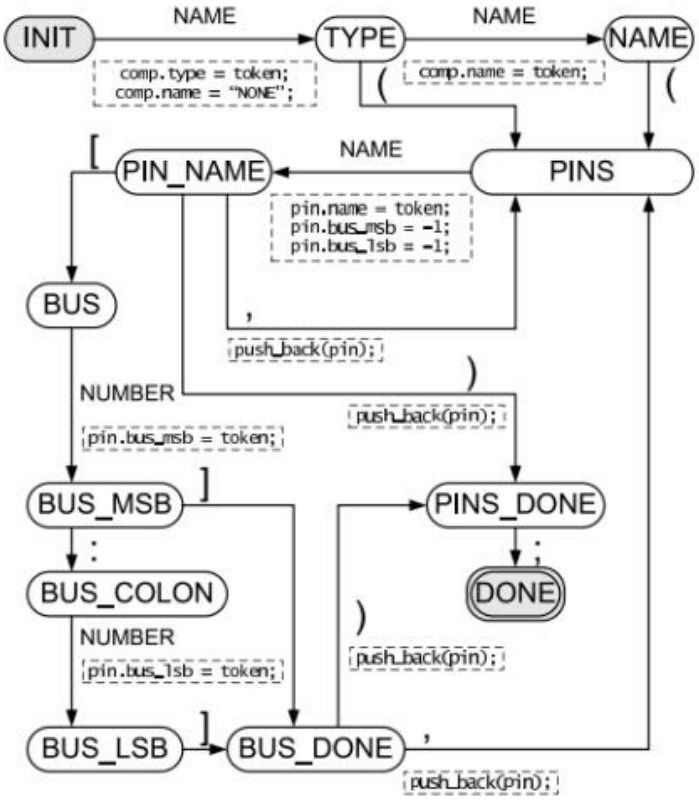

Figure 2. FSM for components

Another key implement is the display statement part. For the required format, I set two global counters to count wire and component numbers and use vector and list visit ways to access them and output the corresponding items. Also, every function in syn.cpp includes exception handling.

4. Test Cases design

In the initial release, I create 5 tests only included different MODULE and ENDMODULE that I implement at that time, and they all pass test and showed in the report.

However, in the final release, I create new 5 tests included all statement types: wire, component, MODULE and ENDMODULE. Every test include MODULE and ENDMODULE, so these two parts are always tested.

Case 1: test.evl

This file will test all types of statement. Key point is that when wire is defined between components, if it will be count and display correctly that .syntax file will show wire and wire count right, then specific wires and rest components. Result: pass.

Case 2: test2.evl

This file will test rightness of components and pins. The language *output sim_out(a[0],s,b[1:0]);* will go through all FSM of components. The .syntax file will display component name, type, number of pins, and detailed of each pin (MSB, LSB and name). Result: pass

Case 3: test3.evl

This file will test rightness of wires. The language *wire [5:0]s0, s1, clk;* will go through all FSM of wires. The .syntax will display wire name and correct wire width. Result: pass

Case 4: test4.evl

This file will test rightness of two different part width wires. The language is *wire [5:0]s0, s1, clk; wire [1:0]a;* The .syntax will display total wire number(4) and then each wire name and correct wire width such as wire s0 6; wire s1 6; wire clk 6; wire a 2. Result: pass

Case 5: test5.evl

This file will test all mixed parts above. Four types and each potential problem I think will be tested. Result: pass

Case 6: golden/cpu32_flat.evl

The file, why I mentioned here because I met a crucial problem about this golden test, has more than ten thousands of lines. And the .tokens file has much more than that. At first time, I use vector to store tokens and statements, and use *erase()* to pop the beginning item, which will cost much more time when iterations reach huge number. I almost ran about 1 hour to finish the test.

Then, I change the vector to list and use *pop_front()* instead of *erase(),* also change access ways to visit list items. Result: response time from almost 1 hour to ten seconds! What horrible and amazing time complexity!