Mini Assignment 1

Akilesh B, CS13B1042 and Phaneendra Babu, CS13B1037

October 7, 2015

Part 1:

Study of C compiler in python

Why C?

- C has a good set of built-in functions and operators.
- C is highly portable. That is, the program once written can be run on other machines with little or no modification.
- It is fast and efficient, modular.

Why python?

- It is easy to code and simple.
- Lots of in-built libraries makes python a suitable language.
- Lexer and Parser is easily coded using support for regular expression matching.

These features of C makes in interesting for compiler writer.

Lexer phase:

- First set of tokens are defined. This includes set of keywords, operators, delimiters.
- keywords = [['int', 'float', 'double', 'char', 'void'], ['if', 'for', 'while', 'do', 'else'], ['include', 'return']]
- operators = ['=', '&', '<', '>', '++', '–', '+', '-', '*', '/', '>=', '<=', '!=']
- delimiters = ['(', ')', '{', '}', '[', ']', ',', '\"', ';']
- There is a class Lexer which performs lexing. Blank characters are defined as ' ' or '\t', '\n', '\r' and whenever it is encountered, we just skip that character and move to the next. (increment index by
- Input which needs to be tokenized is present in content variable. tokens[] contains tokens and it is initialized as empty
- In Lexer, there is a main function tokenizes input. At first, if it is a blank then we skip it.
- It is important to note the priority here. First, it checks if input character is '#' then check if it is followed by "include". If so, we append include to the list of tokens and check what follows include, if '\"' or '<' follows include it is tokenized and included. For all the other cases, we print "include error".

- Then it checks if it is an alphanumeric or '_' or digit or '.' then it matched with an appropriate token defined at the start and next character is encountered.
- Similar actions are performed if it is delimiter or operator.
- Special cases: If it matches with a digit and if somewhere a '.' is encountered we print 'float number error'. In case of delimiter if there is an opening '\"' without a closing '\"' then we print error. In case of operators, cases of '++', '–', '>=', '<=' are handled separately as '+' followed by a '+' will be tokenized as a single '++' and next character we encounter is index + 2.
- The next step is construction of syntax tree. There is a SyntaxTreeNode class which initializes a node's value, type, left, right first_son, extra info etc. SyntaxTree class actually performs the tree construction. The function add_child_node, adds a child node, it determines of the father of the child.

Parsing phase

- Parser first gets the tokens generated by the lexer and constructed Syntax tree is present in tree variable.
- First it checks if it is inside a block (starts with '{'). Inside a block it can be a 'STATEMENT' or 'ASSIGNMENT' or 'FUNCTION_CALL' or 'CONTROL' or 'RETURN'. If a right brace ('}') is encountered we break.
- If it is a 'STATEMENT' then as long as we encounter a ';' we process. statement_tree is initialized to the syntax tree generated.
- The token can be a keyword, identifier, integer constant, nested braces. In each of these cases, it is parsed appropriately and child_node containing details of the variable type , value is added to the statement_tree.
- If it is an 'ASSIGNMENT' then the token can be either an 'IDENTIFIER' or assignment operator.
- If it is a 'FUNCTION_CALL', the token can be an identifier which is trivial. It can also be a recursive function call.
- The statement should end with a semicolon. If not, we print an error.
- The function _expression handles all expressions. At first, operator precedence is specified as follows:
- operator_priority = {'>': 0, '<': 0, '>=': 0, '<=': 0, '+': 1, '-': 1, '*': 2, '/': 2, '++': 3, '–': 3, '!': 3}. That is '!', '++', '–' have highest priority and '>', '<', '>=', '<=' have least priority.

- operator_stack and operand_stack keep track of operators and operands. reverse_polish_expression are initialized as empty list.
- The function _for, _ifelse, _while performs necessary parsing when for, ifelse and while is encountered.
- After parsing is done, Abstract Syntax Tree is constructed. After which semantic analysis and code generation (in assembly language is done).

Unique characteristics of C:

- In C identifiers are case sensitive unlike most other languages.
- There are no nested comments in C.

HTML Lexer and Parsing Analysis:

Why HTML and characteristics which makes it interesting for compiler writer?

- HTML stands for HyperText Mark-Up Language, is the language used for writing web documents pages in the Internet. The language is used for text processing.
- Creating a HTML file is so simple, HTML files are written in ASCII format text, making easy for the user to create his/her web page in a comfortable way.
- HTML is case insensitive with its language commands.
- The characters within the document, however, are case sensitive.
- So lexer matches many words because of case insensitivity but takes care of some key words in the core language.
- The language consists of elemets formed by tags.
- These helps us to understand the the layout, background, headings, titles, text ,images on the file.
- The elements are grouped according to their function in the HTML document. There are head elements and body elements.
- The head elements identify properties of the entire document, while body elements actually mark text as content.
- Most elements have a beginning and an ending which encodes the text the user wishes to mark with the tag. All HTML documents must begin with the element and end with the element.
- Attributes may be used along with the elements. These perform functions such as placement of text, indication of the source files of images, and identification of links to the document or part of the document.

Why ANTLR?

- ANTLR grammars are easy to understand than large codes written for parsing in Java or python. So I chose ANTLR grammmars, it contains direct rules of parsing.

Lexer Phase

- First Glance at the grammar tells us that this this grammar needs recursive rules to match beginning and ending of an entity such as tag,script and style.
- Recursive structures can be defined by using modes by pushing them whenever starting tag is found and popping whenever ending tag is found.
- DTD : '<!' .*? '>'
- This is Document Type declaration and is a formal part of file description
- HTML text has certain constructs called markups. The markup constructs add structure to documents.
- The DTD gives a grammar for the element structure of the specialized markup language: the start symbol is the document element name; the productions are specified in element declarations, and the terminal symbols are start-tags, end-tags, and data characters.
- The lexical analyzer separates the characters of a document into markup and data characters. Markup is separated from data charcters by delimiters.
- The HTML delimiter recognition rules include a certain amount of context information. For example, some token can only be recognized as markup when it is followed by a certain letter.
- HTML_COMMENT : '<!–' .*? '–>' ;
- HTML_CONDITIONAL_COMMENT : '<![' .*? ']>' ;
- Comment declarations and marked section delcarations are other types of markup declarations.
- The string <!– begins a comment declaration. The – begins a comment, which continues until the next occurrence of –. A comment declaration can contain zero or more comments. The string <!> is an empty comment declaration. The string <![begins a marked section declaration.
- CDATA : '<![CDATA[' .*? ']]>'
- The string <! followed by a name begins a markup declaration. The name is followed by parameters and a >.

- TAG_OPEN : '<' -> pushMode(TAG)
- mode TAG; TAG_CLOSE : '>' -> popMode ;
- TAG_SLASH_CLOSE : '/>' -> popMode ;
- TAG_SLASH : '/' ;
- Tags are used to delimit elements. Most elements have a startingtag, some content, and endingtag. Empty elements have only a startingtag. For some elements, the startingtag or endingtag are optional. A startingtag begins with < followed by a name, and ends with >. The name refers to an element declaration in the DTD. An end-tag is similar, but begins with </.
- An attribute value may have spaces b/t the '=' and the value
- ATTVALUE_VALUE : [ ]* ATTRIBUTE -> popMode ;
- ATTRIBUTE : DOUBLE_QUOTE_STRING | SINGLE_QUOTE_STRING | ATTCHARS | HEXCHARS | DECCHARS ;
- Start tags may contain attribute specifications. An attribute specification consists of a name, an "=" and a value specification. The name refers to an item in an ATTRIBUTE declaration. The value can be a name token or an attribute value literal A name token is one or more name characters. An attribute value is a string separated by double-quotes (") or a string separated by single-quotes (').
- If the ATTLIST declaration specifies an list of names, and the value specification is one of those names, the attribute name and "=" may be omitted

Parser phase

- HTML parser rules of HTML are very few in number
- htmlDocument : (scriptlet | SEA_WS)* xml? (scriptlet | SEA_WS)* dtd? (scriptlet | SEA_WS)* htmlElements* ;
- HTML document may or may not contain xml declaration also xml declaration may follow optional scriptlet
- SCRIPTLET : '<?' .*? '?>' | '<%' .*? '%>' ;
- A HTML element could be a script or style or scriptlet or name,attribute value pairs enclosed in tags,ending could be TAG_SLASH_CLOSE or TAG_CLOSE.
- mode TAG; TAG_CLOSE : '>' -> popMode ;
- TAG_SLASH_CLOSE : '/>' -> popMode ;

- A HTML comment could be general comment or conditional comment
- HTML_COMMENT : '<!–' .*? '–>' ;
- HTML_CONDITIONAL_COMMENT : '<![' .*? ']>' ;
- A style could end with either '</style>' or '</>'
- mode STYLE : STYLE_BODY : .*? '</style>' -> popMode ;
- STYLE_SHORT_BODY : .*? '</>' -> popMode ;
- Script also ends similarly to the style.
- HTML character data could be any HTML_TEXT or whitespace or newline.
- htmlChardata : HTML_TEXT | SEA_WS ;
- HTML attribute may or may not contain attribute value but name is compulsory.
- htmlAttribute : htmlAttributeName TAG_EQUALS htmlAttributeValue | htmlAttributeName ;
- A HTML element could be followed and preceeded by any number of whitespaces or newlines.
- htmlElements : htmlMisc* htmlElement htmlMisc* ; htmlMisc : htmlComment | SEA_WS ;

Acknowledgements

- For part 1: https://raw.githubusercontent.com/shiyanhui/Compiler/master/compiler.py
- For part 2: http://www.w3.org/MarkUp/SGML/sgml-lex/sgml-lex for lexical analysis
- ANTLR version4 examples from github.