Capstone Project Report

Student Name: Wenbo Feng
Advisor: Periklis A. Papakonstantinou

2016-12-15

1 Project Description

A grammar for a programming language often appears as a reference for people trying to learn the language syntax. Designers of compilers and interpreters for programming languages often start by obtaining a grammar for the language. Most compilers and interpreters contain a component called a parser that extracts the meaning of a program prior to generating the compiled code or performing the interpreted execution. A number of methodologies facilitate the construction of a parser once a context-free grammar is available. Some tools even automatically generate the parser from the grammar. In this project, our goal is to build a parser for the grammar of OpenScad in order to find out the similar industrial products.

2 Basic Concepts

2.1 Regular Expressions

2.1.1 Regular Expressions in computational theory

In arithmetic, we can use the operations $+$ and $\times$ to build up expressions such as

$$(5 + 3) \times 4.$$

Similarly, we can use the regular operations to build up expressions describing languages, which are called regular expressions. An example is:

$$(0 \cup 1)0 * .$$

The value of the arithmetic expression is the number 32. The value of a regular expression is a language. In this case, the value is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s.

Say that R is a regular expression if R is

1. a for some a in the alphbet $\sum$,
2. ε
3. $\emptyset$
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1*), where R_1 is a regular expression.

2.1.2 Regular Expressions in pragramming language

Regular expressions have an important role in computer science applications. In applications involving text, users may want to search for strings that satisfy certain patterns.Utilities such as awk and grep in UNIX, modern programming languages such as Perl, and text editors all provide mechanisms for the description of patterns by using regular expressions.

The power of the regular expressions comes when we add special characters to the search string that allow us to more precisely control which lines match the string. Adding these special characters to our regular expression allow us to do sophisticated matching and extraction while writing very little code.

some basic examples:

```
print(re.match('www', 'www.runoob.com').span())   #match from the begining

raw_input= "20:07:13.2"
hms_pat = re.compile( r'(\d+):(\d+):(\d+\.?\d*)' )
```

2.2 Context Free Grammar(CFG)

Context-free grammars is a more powerful method of describing languages. Such grammars can describe certain features that have a recursive structure, which makes them useful in a variety of applications.

The collection of languages associated with context-free grammars are called the context-free languages. They include all the regular languages and many additional languages.

The following is an example of a context-free grammar, which we call G1.

$$A \rightarrow 0A1$$
$$A \rightarrow B$$
$$B \rightarrow \#$$

A grammar consists of a collection of substitution rules, also called productions. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a variable. The string consists of variables and other symbols called terminals. The variable symbols often are represented by capital letters. The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols. One variable is designated as the start variable. It usually occurs on the left-hand side of the topmost rule. For example, grammar G1 contains three rules. Grammars variables are A and B, where A is the start variable. Its terminals are 0, 1, and #.

A derivation of string 000#111 in grammar is

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111.$$

3 The Grammar for 3D Modelling

3.1 About OpenSCAD

OpenSCAD is a software for creating solid 3D CAD models. Unlike most free software for creating 3D models (such as Blender) it does not focus on the artistic aspects of 3D modelling but instead on the CAD aspects. Thus it might be the application you are looking for when you are planning to create 3D models of machine parts but pretty sure is not what you are looking for when you are more interested in creating computer-animated movies. OpenSCAD is not an interactive modeller. Instead it is something like a 3D-compiler that reads in a script file that describes the object and renders the 3D model from this script file. This gives you (the designer) full control over the modelling process and enables you to easily change any step in the modelling process or make designs that are defined by configurable parameters.

3.2 OpenSCAD's Grammar

The grammar description can be found at: https://github.com/openscad/openscad/blob/master/src/parser.y. The whole compiler of OpenSCAD is written in lex and yacc, from the lexer.l and parser.y source code, we can extract the grammar of the OpenSCAD's language to build our own parser.

4 Parsing Algorithm

4.1 Dynamic Programming

In a nutshell, dynamic programming is recursion without repetition. Dynamic programming algorithms store the solutions of intermediate subproblems, often but not always in some kind of array or table. In dynamic programming we are not given a dag; the dag is implicit. Its nodes are the subproblems we define, and its edges are the dependencies between the subproblems: if to solve subproblem B we need the answer to subproblem A, then there is a (conceptual) edge from A to B. In this case, A is thought of as a smaller subproblem than Band it will always be smaller, in an obvious sense. As long as we memoize the correct recurrence, an explicit table isnt really necessary, but if the recursion is incorrect, nothing works.

4.2 Earley Algorithm

4.2.1 Description

The algorithm is a bottom-up dynamic programming algorithm with top-down prediction: the algorithm builds up parse trees bottom-up, but the incomplete edges (the predictions) are generated top-down, starting with the start symbol. We need a new data structure: A dotted rule stands for a partially constructed constituent, with the dot indicating how much has already been found and how much is still predicted. Dotted rules are generated from ordinary grammar rules. The algorithm maintains sets of states, one set for each position in the input string (starting from 0).

4.2.2 Algorithm

The Earley algorithm has three main operations:
Predictor: an incompleteentry looks for a symbol to the right of its dot. if there is no matching symbol in the chart, one is predicted by adding all matching rules with an initial dot.
Scanner: an incompleteentry looks for a symbol to the right of the dot. this prediction is compared to the input, and a complete entry is added to the chart if it matches.
Completer: a completeedge is combined with an incomplete entry that is looking for it to form another complete entry. The pseudocode is as follows

5 lex and yacc

5.1 lex

5.1.1 introduction

The program Lex generates a so called 'Lexer'. This is a function that takes a stream of characters as its input, and whenever it sees a group of characters that match a key, takes a certain action. A very simple example:

```
%{
#include <stdio.h>
%}
%%
[0123456789]+ printf("NUMBER\n");
[a-zA-Z][a-zA-Z0-9]* printf("WORD\n");
%%
```

5.1.2 An example

Lets say we want to parse a file that looks like this:

```
logging {
          category lame-servers { null; };
          category cname { null; };
};
zone "." {
          type hint;
          file "/etc/bind/db.root";
};
```

We clearly see a number of categories (tokens) in this file:

- WORDs, like 'zone' and 'type'
- FILENAMEs, like /etc/bind/db.root
- QUOTEs, like those surrounding the filename

Algorithm 1 Earley-Parser

```
function INIT(words)
    $S \leftarrow$ CREATE-ARRAY(LENGTH(words))
    for $k \leftarrow$ from 0 to length(words)] do
        $S[k] \leftarrow$ EMPTY-ORDERED-SET
    end for
end function
function EARLEY-PARSE(words, grammar)
    $INIT(words)$
    ADD-TO-SET$((\Upsilon \leftarrow \bullet S, 0), S[0]))$
    for $k \leftarrow$ from 0 to length(words)] do
        for $k \leftarrow$ from 0 to length(words)] do
            if not FINISHED(state) then then
                if $NEXT - ELEMENT - OF(state) isanonterminalthen$ then
                    PREDICTOR(state, k, grammar)
                else
                    CANNER(state, k, words)
                end if
            else
                COMPLETER(state, k)
            end if
        end for
    end for
    return $chart$
end function
function PREDICTOR(($A \leftarrow \alpha \bullet B\beta$, j), k, grammar)
    for each ( $B \leftarrow \Upsilon$) in GRAMMAR-RULES-FOR(B, grammar) do  do
        ADD-TO-SET(( $B \leftarrow \bullet\Upsilon$, k ), S[k])
    end for
end function
function PREDICTOR(($A \leftarrow \alpha \bullet a\beta$, j), k, words)
    if a then $\subset$ PARTS-OF-SPEECH(words[k]) then
        ADD-TO-SET(( $A \leftarrow \alpha a \bullet \beta$, k ), S[k+1])
    end if
end function
function COMPLETER((B $\leftarrow \Upsilon\bullet$, x ), k)
    for each ($A \leftarrow \alpha \bullet B\beta$, j) in S[x] do  do
        ADD-TO-SET( $A \leftarrow \alpha B \bullet \beta$, j ), S[k])
    end for
end function
```

- OBRACEs, {
- EBRACEs, }
- SEMICOLONs, ;

The corresponding Lex file is Example 3:

```
%{
#include <stdio.h>
%}
%%
[a-zA-Z][a-zA-Z0-9]*                 printf("WORD ");
[a-zA-Z0-9\/.-]+                     printf("FILENAME ");
\"                                   printf("QUOTE ");
\{                                   printf("OBRACE ");
\}                                   printf("EBRACE ");
;                                    printf("SEMICOLON ");
\n                                   printf("\n");
[ \t]+                              /* ignore whitespace */;
%%
```

When we feed our file to the program this Lex file generates (using example3.compile), we get:

```
WORD OBRACE
WORD FILENAME OBRACE WORD SEMICOLON EBRACE SEMICOLON
WORD WORD OBRACE WORD SEMICOLON EBRACE SEMICOLON
EBRACE SEMICOLON
WORD QUOTE FILENAME QUOTE OBRACE
WORD WORD SEMICOLON
WORD QUOTE FILENAME QUOTE SEMICOLON
EBRACE SEMICOLON
```

When compared with the configuration file mentioned above, it is clear that we have neatly Tokenized it. Each part of the configuration file has been matched, and converted into a token.
And this is exactly what we need to put YACC to good use.

5.1.3 PLY (Python Lex-Yacc)

PLY is a pure-Python implementation of the popular compiler construction tools lex and yacc. PLY consists of two separate modules; lex.py and yacc.py, both of which are found in a Python package called ply. The lex.py module is used to break input text into a collection of tokens specified by a collection of regular expression rules. yacc.py is used to recognize language syntax that has been specified in the form of a context free grammar. In our project, we just take advantage of lex.py as our lexical scanner tool to tokenize the input file.

The two tools are meant to work together. Specifically, lex.py provides an external interface in the form of a token() function that returns the next valid token on the input stream. yacc.py calls this repeatedly to retrieve tokens and invoke grammar rules. Unlike traditional lex/yacc which require a special input file that is converted into a separate source file, the specifications given to PLY are valid Python programs. This means that there are no extra source files nor is there a special compiler construction step (e.g., running yacc to generate Python code for the compiler). Since the generation of the parsing tables is relatively expensive, PLY caches the results and saves them to a file. If no changes are detected in the input source, the tables are read from the cache. Otherwise, they are regenerated.

5.2 Yacc

5.2.1 Introduction

YACC can parse input streams consisting of tokens with certain values. This clearly describes the relation YACC has with Lex, YACC has no idea what input streams are, it needs preprocessed tokens. While you can write your own Tokenizer, we will leave that entirely up to Lex. A note on grammars and parsers. When YACC saw the light of day, the tool was used to parse input files for compilers: programs. Programs written in a programming language for computers are typically *not* ambiguous - they have just one meaning. As such, YACC does not cope with ambiguity and will complain about shift/reduce or reduce/reduce conflicts.

5.2.2 Parsing a configuration file

Lets repeat part of the configuration file we mentioned earlier:

```
zone "." {
          type hint;
          file "/etc/bind/db.root";
};
```

Remember that we already wrote a Lexer for this file. Now all we need to do is write the YACC grammar, and modify the Lexer so it returns values in a format YACC can understand.
In the lexer from the lex example we see:

5.2.3 The Parser in this project

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
zone return ZONETOK;
file return FILETOK;
[a-zA-Z][a-zA-Z0-9]* yylval=strdup(yytext); return WORD;
[a-zA-Z0-9\/.-]+ yylval=strdup(yytext); return FILENAME;
\" return QUOTE;
\{ return OBRACE;
\} return EBRACE;
; return SEMICOLON;
\n /* ignore EOL */;
[ \t]+ /* ignore whitespace */;
%%
```

The grammar:

```
commands:
|
commands command SEMICOLON
;

command:
zone_set
;

zone_set:
ZONETOK quotedname zonecontent
{
printf("Complete zone for '%s' found\n",$2);
}
;
zonecontent:
OBRACE zonestatements EBRACE

quotedname:
```

```
QUOTE FILENAME QUOTE
{
$$=$2;
}

zonestatements:
|
zonestatements zonestatement SEMICOLON
;

zonestatement:
statements
|
FILETOK quotedname
{
printf("A zonefile name '%s' was encountered\n", $2);
}
;

block:
OBRACE zonestatements EBRACE SEMICOLON
;
statements:
| statements statement
;

statement: WORD | block | quotedname
```