Self Introduction:

I am a freshman in Tsinghua University, Institute for Interdisciplinary Information Sciences. This is actually my first time formally start to learn Computer Science. I used to focus on Physics and Mathematics but in university I decided to do I something new and “practical”. I am very interested in algorithms and cool applications, techniques, research fields… almost everything on computers. Although I started in this major late, I really love it.

Selfie: selfie.jpg

Reasons why I choose this lesson:

At first glance, I understand that in this class I will learn how a computer is composed from bottom to top. But I really can’t understand what is computational thinking. I thought it might be some thinking strategies for new knowledge or solving problems. Anyway, it sounds really exciting and cool so I chose it.

Feeling right now:

To be honest, this might be the best class I take this semester. I found the concept of “Modularity”, “Abstraction”, “Staging” and “Focus” very profound and meaningful when I was doing some personal projects. Moreover, the teacher introduced lots of modern approaches to co-work, edition control. I could see almost the newest stuff in the computer world. It makes feel like an actual cool geek LOL.

Logic Model:

|  |  |
| --- | --- |
| Background | In Chapter 9 we have learned Jack language, which is designed for human’s expressing habits. In Chapter 10, we need to tokenize and parse Jack code to be prepared for the translation. However, you will discover much more fun in implementing your ideas. |
| Target | Get a deeper understanding of \*how\* and \*why\* we need to tokenize and parse and furthermore understand how powerful “Abstraction” and “Staging” are. |
| Input | Nand2tetris materials (design contract and test files), online docs about regular expressions, programming tutorial (optional). |
| Output | A compiler that can pass all the homework. A small program that can talk in natural language (optional). |
| Progress | * Figure out what’s tokenizing and parsing * Learn how to write regular expressions * Learn to use Lex and implement functions with C code * Try using ANTLR or Yacc to accomplish the same task * Try use the similar “Abstraction” and “Staging” to do natural language processing |
| Effect | * Can use regular expressions and corresponding software to do tokenizing and parsing * Know the cons and pros of the two ways to achieve the same target. * Understand why the book recommend us to take this two-step method. * Have a sense of how important and pragmatic “Abstraction” and “Staging” can be * Find the similarity in analyzing different languages and use that to do natural language processing |

= 10.0 Introduction=

Basically in this chapter, we going to do tokenizing and parsing.

\*Tokenize

\*\*Definition: It's the process of demarcating and possibly classifying sections of a string of input characters(From Wikipedia-Tokenization). In Natural Language Processing is like turning "Hello World !" in to "Hello", "World" ,"!". When we meet a more meaningful sentence like "You are smart.", we should do "You"(noun), "are"(verb), "smart"(adjective).

\*\*In Nand2tetris: When we try to do natural language processing, it's not that easy since our grammars are much more complicated and sometime ambiguous. Imagine tokenize "Can your can a can ?". However, you don't need to worry too much about tokenizing Jack Language because all the grammars and vocabulary are designed to be clear and certainly unambiguous. For example, "let x = y " should be tokenized as "let"(keyword), "x"(identifier), "="(symbol), "y"(identifier). And of course, identifiers should not be names as "let", "if"...... So basically, there is a table in the Nand2tetris book to tell you exactly how those words are distinguished.

\*\*Example: tokenizeinput.jpg tokenize.jpg

\*Parse

\*\*Definition: The process of analyzing a string of symbols, conforming to the rules of a formal grammar.(From Wikipedia-Parsing). So instead of just recognizing every single word, this time we should distinguish the whole structure. For example, when doing "I catch you with my hands.", you should figure out the main structure is "I"(subjective) "catch"(verb) "you"(objective). and "with hands" is some kind of adverb.

\*\*In Nand2tetris: Somehow the structure in Jack Language may be complicated. We have classes, functions, methods, declaring variables and so on. However, since all the structures are also defined well, there will certainly be no confusion when parsing. You can figure out those structures in Book Chapter 9 and 10.

\*\*Example: parseinput.jpg parse.jpg

=10.1 Background=

\*Why we need to tokenize and parse?

\*\*The most straightforward reason is that we need to write a compiler this time. You may say that in the previous chapters we have already done several compilers. That's true since basically, we can do all things with lots of gates. But directly manipulating gates seems like the worst idea because of its complexity. So we got HDL -> machine language -> assembly language -> virtual machine language.

\*\*However, virtual machine language works well for machines but still seems complicated for people to use. Also, you may need to worry about too many details when using virtual machine language. So Jack language is introduced in Chapter 9 as a programming language, which should be understandable and kind of native to human. And I guess the basic aim of programming languages is to let people use them as speaking or writing in mother language. So the difference this time is we actually have more complicated sentences and semantics, which makes it much more difficult to translate.

\*Okay, but why two steps?

\*\*It's clear that our ultimate goal is to turn Jack language into virtual machine language so that everything feasible in Jack language can be complied downwards through several layers and work well on machines. So how to do it is actually up to you. To do it in two steps is highly recommended. However, we have also done it in a one-step bitter way so you can find out why two-step is more recommended.

=10.2 How do we tokenize=

==10.2.1 Regular Expression==

You can find the concept of it here Regular Expression from Wikipedia. Basically, we use regular expressions to "filter". We can see a quick example here:

[a-zA-Z0-9] and input "Area = 3.14 \* r ^ 2" // Here what we want to do is to "filter" all the identifiers.

After the filtering, you can get "Area" and "r".

Ideally, we want to just filter the input text and output the corresponding token. But we are now still a few steps further to go.

==10.2.2 Lex==

'''Here are some major problems we may face in tokenizing:'''

\*Deal with the ambiguity

\*\*For example, we don't want to recognize "let", "if" or "function" as an identifier. And the way to fix that is the priority of regular expressions.

Deal with different encoding

\*\*Chances are that the regular expressions you write are supposed to recognize letters in uft-8, but the input text isn't in that format. Or you may set white space's regular expressions as ' ', but there are actually '\t', '\n' and maybe other forms. According to my experience, I would recommend you to try to include all the possible cases in regular expressions and turn whatever input you get into utf-8, and then do the work.

'''What Lex can do:'''

\*Concept:

\*\*It's a computer program that generates lexical analyzers.

\*Structure

\*\*Definition section: Here you can include all the head files you need, declare and initialize all the variables you use later. Basically, you can write any C codes here.

\*\*Rules section: You can write your regular expressions here, with corresponding operations.

\*\*Example: lexquickexample.jpg

\*How it works

\*\*The program will scan the input text only once from start to end. So from the first letter, it will try to find the matching regular expression. If it finds a feasible one (or maybe more, we will talk about this later), it just stops to execute the corresponding operations and then try to match again from where it stops. If it doesn't find any match, it will just print the not-matched letter, and try to match from the next letter.

\*How it fix ambiguity

*\*First, for a given string, the regular expression which matches longer wins. For example, "let" beats [a-z]* when recognizing the "let" in "let x = 1" because "let" matches 3 letters while [a-z]\* only matches the first letter in regular expressions.

\*\*Second, if two regular expressions have the same number of matching letters, the one written first in Lex wins. So you may want to write "let", "if", "class" and other keywords, symbols at first, and identifiers, numbers at last.

==10.2.3 Implementation==

The details of how to write Lex and run it on different operating systems can be easily found on Google. And there aren't too much to say in tokenizing because we are simply just writing some rules and directly print what we scan. Just remember dealing with white spaces, pay attention to the encoding format and it should be good.

However, if you meet any difficulty, you can check our Github repository and find the Lex code there.

=10.3 How do we parse (in Lex)=

==10.3.1 What's different about parsing==

\*Format: We have indentation now. You can see from the examples provided in the book or from the website that there exists some kind of structure in the output xml file. For example, when we compile a function, the body of the functions such as declaring variables or doing statements appear to belong to the function, like this (You can see those “subroutine”s): parsexml.jpg

\*Context: When we tokenize, every word generates a single token and we are done. But now, one word may generate several lines and have relations with context. For example, when we compile '-', if it means negative like "-3", the output should be "<term><symbol>-</symbol><Number>3</Number></term>". But if it means minus, the output will be "<term>...</term><symbol>-</symbol><term>...</term>"

==10.3.2 My original idea==

Okay, now we have done tokenizing perfectly. Although parsing seems to be much complicated, I noticed that there is additional storage space in Lex, which means we can basically implement whatever we can do in C in Lex. And by the storage space and the priority of regular expressions, I find that I might be able to parse directly without any tokenizing.

I will show what I did, and you should notice that it's not "right" although it runs right.

'''Main difficulties:''' I think the biggest difficulty here is that we can just execute one set of operations (just use '{' and '}' to include all the operations you want) for each regular expression. So we must judge lots of things. For example, when we get '}', it may be the end of a function, a class, an if statement, an else statement or a while statement and the output varies. Also when we see arithmetic expressions like "2(-3+x)-y", we have to give a good structure of all the terms in it. Here ')' can be the end of an arithmetic expression, the parameter list of a function and so on.

My solution is to use some public variables and stack.

==10.3.3 Stack==

You sure have learned this data structure in the previous chapters. What it achieves is a first in last out structure.

Let's still consider the '}' case. We can create a stack called A and when we scan through the whole input text, we push '1' for "class", '2' for "function", '3' for "if", '4' for "while"...... Every time we encounter a '}', we just pop from A, see what number we get, output the corresponding end statement.

And for every ambiguous symbol, we create a stack for it, which should deal with the context problem.

==10.3.4 Implementation==

\*Remember to put all the necessary functions in a head file so that you can keep your code neat.

\*When you are trying to use the stack, you can simply implement with arrays. For example, you may declare int a[10]; in the declaration part. And write your push, pop function with the index of the temporary top element recorded every time (You may want to check the details in my code in Github).

\*In some cases you may keep resetting values. For example, my strategy for judging "-" as minus or negative is to set a variable called num, which is initialized to 0. Every time we scan a "(" or "=" we set num to 1. Every time we scan a number or ")", we set num to 0. So we can judge the status of "-" by simply reading num.

=10.4 How to parse in ANTLR (Yacc)=

==10.4.1 What we have achieved now==

\*Well, if you have implemented those codes in Lex, then we already can do all the tokenizing and parsing. Actually, when we parse, we still use basic regular expressions to filter and some storage space and data structure to handle the context problems.

\*However, if you have really finished those codes, or even maybe just Ace the first homework, you must have found that writing C codes in Lex is not a delightful job, especially you have to do judges, value settings and so on. My head file is like 5 hundred of lines. For sure a compiler must be difficult to write, but I ask myself is that my best?

==10.4.2 Bigger regular expressions==

\*Remember when we were doing tokenizing, things are wonderful and really simple. Can't we just keep that when parsing? The answer is "Yes, we can."

\*When we are using stacks, setting public variables to indicate the status of ambiguous letters like "-", "}", what we are doing is basically trying to express some bigger rules or patterns. We already know that "}" must end some "if", "function" statements. We already know that "-" must follows some symbols or leading some numbers or identifiers. We are just trying to express those already known patterns, in an inelegant way (I mean our code must be messy). But remember regular expressions are also expressing patterns! What if we can write a bigger regular expression, which takes tokens as input?

\*That's exactly what ANTLR (or Yacc, they are pretty much the same thing) does. Let's see a very straightforward example about handling arithmetic expressions (its mechanism is very similar to parse): antlr.jpg antlroutput.jpg

\*So what we need to do now is just write some bigger regular expressions and debug a few times and have a cup of coffee or some snacks, and the compiler is done.

==10.4.2 Cons and pros==

\*Lex:

\*\*It's definitely fast, because we just scan once, and all the parsing is done. We have good data structures and strategies to handle different contexts.

\*\*But!!! It's difficult to debug because we didn't '''Abstract''' enough!!! Imagine some new grammars are added or the existing grammars are changed a little bit then the code must be greatly modified.

\*ANTLR (Yacc):

\*\*The main idea is very straightforward——regular expressions are our '''Abstraction''' for tokenizing and parsing patterns. We just write down our known patterns, and we are done.

\*\*Also it's very easy to debug since we have established very good and clear stages of the process. If the grammar changes, we simply modify a little bit on the parsing regular expressions. And any new tokens or grammars are just a few new lines.

\*\*Most importantly, Jack is a relatively simple language. When we try to parse HTML or other programming languages, there are far more context problems. You do not want to write explicit and long long long C code for every situation.

==10.4.3 Depth-first Search==

\*You may be interested in how regular expressions work. You can just google depth-first search and for regular expressions, it's just trying to match patterns one after another.

\*And by knowing that, you can deliberately arrange the orders of regular expressions to make the scanning more efficient.

=10.5 Natural Language Processing=

\*Since Machine learning is extra-ordinary hot these days and I was also studying it, I would like to introduce some of my thoughts here (I'm really not an expert LOL):

\*\*Basic Processing: We now know how to do tokenizing and parsing to Jack Language, I believe we can also do that to the natural language since the '''Abstract''' is totally the same! When we are talking or writing, words are just tokens, and all the meaningful sentences must have some existing patterns like "subjective-verb-objective". We may download data from a dictionary so we get every word's possible properties (Maybe they should be ranked by frequencies? ). We may need to learn a few grammar lessons to write more regular expressions. Then I believe we are able to handle every input sentence and store all the events.

\*\*Example: "I (Toby) appreciate your (Jack) reading this book. " This sentence is quite easy to handle, and the output with pre-stored data could be like nlp.jpg

\*\*If we have enough common sense processed and stored like this and we are able to process conversations immediately, we may write some patterns for questions and we are able to talk to other people artificially!!

\*\*Possible application: Most importantly, we are talking with memorizing! If we can import some neural networks, train some models so we can to some extent handle ambiguous sentences so we are able to grow when talking and talk better in the future.