**KoreanParser docs & spec**

<https://github.com/GoldenMine0502/KoreanParser5>

위 프로그램은 한국어를 루트 문법으로 하는 한국어 기반 프로그래밍 언어입니다.

인터프리터 언어이며 kotlin으로 구성되어 있습니다.

MainTest에 테스트 케이스로 여러 예제 코드를 적어 두었으니 참고하시기를 바랍니다.

**개발의 발단 (썰, 안읽어도 됨)**

초등학교때부터 뭔가 프로그램을 만들어보고 싶었지만 어렵다는 선입견 때문에 접근조차 하지 못했던 적이 있었다.

중학교 1학년 때 스크래치를 접하면서 선입견을 깰 수 있게 되었고 이후 C언어를 공부하였고 자바를 공부하면서 실력을 올릴 수 있게 되었다.

아무래도 접근하기 편하고 영어보단 네이티브인 한국어로, 더 쉬운 문법으로 만든다면 나 같은 사람이 조금이라도 선입견을 쉽게 깰 수 있지 않을까 하는 생각으로 개발하였다.

대단한 것 중 하나는 자연어 처리도 모를 시절 무모한 도전을 무려 4년동안이나 지속했다. (물론 중간에 공백도 있고 지금도 자연어 처리 수업을 들어본 적은 없음)

개인적으로 반성하는 것으로 차라리 수업 또는 학습이라도 좀 하고 했으면 개발 시간이 많이 줄었을 거라고 생각한다. 물론 이렇게 해서 장점이 없는 것도 아니다. 이 프로그램을 짬으로서 내 개발 실력이 엄청나게 올라갔다는 것만큼은 사실이다. 그래도 앞으로는 관련 이론이라도 보고 짜는게 좋겠다고 생각한다.

**프로그램 요약**

1. 실제 한국어와 비슷한 맥락으로 프로그래밍 가능
2. 추상화로 언제든지 lib의 폭을 넓힐 수 있음. 즉 새로운 문장을 적용할 수 있음.
3. 인터프리터 언어
4. Kotlin(JVM) 기반

**라이브러리는 얼마나 사용했는가**

자연어 처리는 크게 형태소 분석(조사), 형태소 분석(어미), 품사 부착, 구 단위 분석, 절 단위 분석이 있다.

형태소 분석(조사) – 직접 구현

형태소 분석(어미) – komoran 라이브러리 사용

품사 부착 – 직접 구현

구 단위 분석 – 직접 구현

절 단위 분석 – 직접 구현(개발 중)

**한국어를 분석하는 과정**

codeProcessor.compile(  
 new OriginalBackupParser(  
 new GenitiveParser(//new SentenceMultiDataParser(  
 new PronounParser(  
 new VariableConnectorParser(  
 new SentenceLastParser(  
 new SentencePastParser(//new BoeoParser(  
 new DefaultParser(null)  
 )  
 )  
 )  
 )  
 )  
 )  
 //))  
 , null);

codeProcessor.compile(new SentenceMultiDataParser(null), null);

현재로서는 총 8단계로 나뉘게 된다.

**DefaultParser**

가장 파싱의 기본이 되는 파서이다.

먼저 위 프로그램에선 전제 조건을 단다.

1. 사용자는 조사 뒤에서 띄어쓰기를 명확하게 할 것.

몇 가지 전제 조건이 필요할 수 밖에 없는 이유는 애초에 자연어라는 것이 예외가 매우 많고 문맥의 약간의 틀어짐이 문장 전체를 좌지우지 할 수 있다는 것이다.

이러한 문제 때문에 프로그래밍 언어 특성상 “명확”해야 하므로 관용적 표현이나 문맥을 파악해야 하는 문법 요소는 모두 배제하였다. 즉 띄어쓰기를 하지 않아도 사람이 문장을 보는 데에 크게 무리가 없지만 이는 문맥적 요소이므로 배제하였다.

lib을 사용할 때도 문맥 전체를 인식해 결과를 바꾸지 않도록 낱말 이상으로 분석하려고 하지 않았다.

위 파서는 가장 기본적인 일을 한다. 스페이스바의 앞에 있는 형태소 몇 개를 인식한다.

예를 들어, “철수가 학교에 갔다”라는 문장이 있다면 스페이스바 앞에 있는 ‘가’, ‘에’ 같은 형태소들을 인식한다는 것이다. 그리고 ‘가’, ‘에’에는 각각 주격 조사, 부사격 조사라는 꼬리표를 붙여준다.

DefaultParser는 문장을 쪼개 역할을 붙여주는 역할을 기본적으로 하게 된다.

ParseContext라는 객체가 위 파서들의 통합 모델 역할을 한다. 모든 데이터를 이 객체에 저장한다.

**SentencePastParser**

parsed.*forEach* **{** if(**it** != null) {  
 if (it.isString() && PronounStorage.INSTANCE.isPronoun(**it**.stringValue())) {  
 depth++  
 }  
 }  
**}**

각 서술어에 대해 depth를 부여한다. 안긴 문장이 얼마나 안겨있는 지 대략적으로 파악하게 된다.

예를 들어, “A에 B를 더한 것을 C에 저장합니다”와 같은 문장이 있다면 안겨 있는 ‘더한’은 1, 가장 밖에 있는 ‘저장하다’는 0이 된다. 기준은 대명사(Pronoun)가 한다.

안은 문장을 인식하는 것은 관형적 수식 단 한 가지로 가능하도록 제약을 걸었으므로 위와 같은 파싱이 가능해진다.

대명사의 종류로는 것, 나머지, 때 등이 있다.

A에 B를 더한 것에 C를 뺍니다

더한 = depth 1

빼 = depth 0

**SentenceLastParser**

sentenceToIndices.*forEach* **{**// *TODO ㄷ* val nedcon = additionals.contains(**it**.second)  
 val selcon = selectables.contains(**it**.second)  
 if(addSet.size >= neededs.size + selectables.size)  
 return@forEach  
  
 if(**it**.second != "서술어" && !addSet.contains(**it**.second)) {  
 if(debug)  
 *println*("SentenceParserUtil-subsequent: ${**it**.first} ${**it**.second} ${additionals.size} ${selectables.size}")  
 addSet.add(**it**.second)  
 if(additionals.size > 0) {  
 subMap[additionals.removeAt(0)] = **it**.first  
 } else if(selectables.size > 0) {  
 subMap[selectables.removeAt(0)] = **it**.first  
 }  
 }  
// if(additionals.size == addSet.size) {  
// return@forEach  
// }  
 **}**

가장 복잡한 파서이다. 요악하면 DefaultParser로 추출한 서술어에 추출한 각종 문장 성분들을 서술어에 붙여주는 역할을 한다. 즉 문장을 연결하는 파서이다.

위 프로그램에는 서술어에 대해 붙어오는 문장 성분을 입력해야 한다.

예를 들어, ”더하다” 라는 서술어가 있으면 부사격에, 목적어가 와야 한다.

이러한 부사격에, 목적어를 서술어의 앞에서부터 찾아내고, 이를 서술어에 최종적으로 붙여주게 된다.

이 외에 문장 인식률을 높이기 위해 작은 코드 조각들이 많이 있지만 해당 독스에선 생략하였다.,

**VariableConnectorParser**

fun parseVariableAndConnector(context: Context): MutableList<Variable?> {  
 val 접속조사리스트 = JosaStorage.getJosaList(JosaCommunity.접속조사)  
 val verify = IParser.PostPositionVerify(접속조사리스트[0], 0)  
  
 var result: List<Context>? = *defaultParse*(context.source, 접속조사리스트, true, verify)["접속조사"]  
  
 if (result == null) {  
 result = ArrayList()  
 }  
 //println(result)  
 return result.stream().map **{** *parseVariable*(**it**.source) **}**.collect(Collectors.toList())  
}

fun parseVariable(str: String): Variable {  
 if (str.*startsWith*("\"") && str.*endsWith*("\"") || str.*startsWith*("\'") && str.*endsWith*("\'")) {  
 return Variable(str.*substring*(1, str.length - 1), false)  
 }  
 if (*isBoolean*(str)) {  
 return Variable(str.*toBoolean*(), false)  
 }  
 if (*isInteger*(str)) {  
 return Variable(str.*toLong*(), false)  
 }  
 return if (*isDouble*(str)) {  
 Variable(str.*toDouble*(), false)  
 } else Variable(str, false)  
  
}

앞 SentenseParser들이 격조사로 큰 틀로 나누어줬다면, 위 파서는 접속 조사를 잘라내고 상수를처리하는 일을 한다.

“A에 1과 2를 더합니다” 같은 문장이 있다면 ‘1과 2’ 라는 것을 1, 2로 나누게 되고 나눠진 1, 2를 상수로 처리한다.

**PronounParser**

대명사와 대명사를 수식하는 서술어를 연결하는 파서이다.

**GenitiveParser**

아직 개발중인 기능이다. ‘의’에 대해 파싱하는 파서이다.

현재 객체지향 관련 기능을 개발 중이고 ‘의’를 통해 더 많은 기능을 제공할 예정이다.

Vector라는 클래스가 있다면 x좌표,y좌표,z좌표라는 변수가 안에 들어 있을 것이고

Vector를 A 변수에 넣었다고 가정 시

[A]의 x좌표를 1로 설정합니다

와 같은 문법을 만들 예정이다. 현재로선 완벽하지 않다.

**OriginalBackupParser**

현재 라인에 대한 초기 정보를 저장한다.

라인이 지나면 이전 데이터로 롤백하게 된다.

차후 최적화를 통해 개선할 예정이다.

**SentenceMultiDataParser**

If문, while문의 경우는 다중 라인 인식이 필요하다.

자바의 경우 if() { 로 열었으면 반드시 뒤에 }가 나와야 한다.

이러한 쌍을 찾아주는 파서이다.