**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 참고되시기를 바랍니다.

초기 마인크래프트 기반으로 구현하였지만 현재는 마인크래프트를 지원하지 않고 직접 동작합니다.

**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.

스크래치로는 뭔가 화면에 보이는 게 있었고 한국어라 친숙했었다. 그래서 선입견을 깰 수 있었다고 생각한다.

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

내가 생각하는 대단한 것 중 하나는 자연어 처리도 모를 시절 무모한 도전을 4번을 엎으며 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 짜는 게 좋겠다고 생각한다.

이 프로그램을 구현하기엔 너무 어려워서 웬만하면 버그를 고치다가 다른 버그가 터지는 불상사를 안 일으킬 때까지 많은 시간 훈련하였지만 위 프로젝트는 간혹 그 불상사가 일어났다. 그 때 마다 코드의 논리를 다시 한 번씩 살피면서 힘들지만 고쳐나갔다. 그렇지만 아직도 버그는 많을 것이다.

**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)

처음에는 어미 형태소도 직접 구현하려 시도하며 낱말 초성 중성 종성 분리하기 등등 다양한 시도를 하였지만 어미의 종류가 너무 다양하여 결국 라이브러리를 사용하였다.

Komoran을 사용함으로써 ~합니다 체만 인식할 수 있던 기존의 한계를 넘어 ~하다 ~해라 등등 다양한 어미도 인식할 수 있게 되었다.

최근에 안 사실로 어미를 분석하기 위한 알고리즘이 따로 존재한다고 한다. 이를 통하여 직접 구현해보는 것도 시도해볼 예정이다.

**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.

가장 초기 버전에 개발했던 파서이다. 위 파서로는 안은 문장을 전혀 인식할 수 없었다.

즉 java로 치면 method(parameters) 이상으로 불가능했던 것이다.

**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 부사어(에), 목적어가 와야 한다.

이러한 부사어(에), 목적어를 서술어의 앞에서부터 찾아내고, 이를 서술어에 최종적으로 붙여주게 된다.

이 외에 문장 인식률을 높이기 위해 작은 코드 조각들이 더 있다.

처음에는 단순하게 문장을 적으면 맨 뒷문장의 낱말에 따라 앞 문장 성분을 지정된 함수로 전달하는 수준에 불과하였다. 하지만 이는 안은 문장을 인식할 수 없다는 문제에 직면하였다.

따라서 대명사라는 개념을 추가하게 되면서 SentenceParser가 필요해지게 되었다.

대명사를 추가하고 이를 수식하는 개념을 도입하였다.

대명사는 특정한 종류로 한정된다: 것, 나머지 등

이러한 단어를 발견하게 되면 대명사에서 가장 가까운 서술어를 수식하는 문장으로 입력 받는다.

이 때부터 method(anotherMethod(parameters), parameters)가 가능해졌다.

**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.

파라미터를 입력할 수 있는 폭이 넓어졌다.

예전에는 method(value1, value2)만 가능했다면 지금은 method(list(value1, value2), value3)이 가능해졌다.

**PronounParser**

대명사와 대명사를 수식하는 서술어를 연결하는 파서이다.

**GenitiveParser**

‘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 설정합니다와 같은 문법이 가능하다.

위 파서를 통하여

Vector3d v3d = new Vector3d(1, 1, 1); v3d.x = 3;

이 가능해졌다.

다만 변수 플레이스 홀더에 대해 아직 완벽하지는 못하다.

**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.

따라서 if, while을 구현할 수 있게 됐다.

**코드를 통한 파싱 논리**

최대한 한국어의 고증에 맞게 설계하기 위해 노력했습니다.

test/java/kr/goldenmine/MainTest.java에서 소스코드를 확인하고 테스트해볼 수 있습니다.

(test1)

***[A]를 1로 설정합니다***

[ ]으로 감싸져 있으면 그것은 변수라는 의미입니다(php에서 $A와 같은 의미).

즉 A에 1을 넣는 것이 됩니다.

서술어 “설정합니다” 에는 목적어(을,를), 부사어(으로,로)를 입력받도록 미리 설정해 두었습니다.

이렇게 하면 VariableStorage의 HashMap에 A에는 1을 집어 넣게 됩니다.

설정하지 않은 변수에 접근하면 무조건 0이 출력됩니다.

***[A]를 출력합니다***

[A]의 값을 출력하는 구문입니다.

“출력합니다”에는 목적어가 들어가 있습니다.

***[A]를 출력하세요***

***[A]를 출력해라***

***[A]를 출력해주세요***

서술어의 어근만 맞아 떨어진다면 어미가 어떤 것이 오던 상관이 없습니다.

(test2)

***[A]가 [B]와 같다***

***[A]와 [B]가 같다***

여타 언어에서의 조건문과 같습니다.

하지만 같다 자체는 함수이며 true 또는 false를 반환합니다.

[A]의 값과 [B]의 값이 일치하다면 true를 반환합니다.

***[A]가 [B]와 같다면***

***그렇지 않다면***

***조건문의 끝***

마지막 서술어의 어미가 “라면”, “ㄴ다면”, “다면”, “자면”, “면”, “으면” 으로 끝난다면 이는 조건문으로 판단하게 됩니다.

앞 서술어의 결과가 true냐 false냐에 따라 if문의 기능을 수행합니다.

if문이 true라면 첫줄~”그렇지 않다면” 까지 실행하게 되고 false라면 “그렇지 않다면”~”조건문의 끝”까지 실행하게 됩니다.

“그렇지 않다면”, “조건문의 끝”은 따로 지정한 상수와 같은 것입니다. Else, endif를 의미합니다.

“그렇지 않다면”은 생략 가능합니다.

단, 조건문의 끝은 생략할 수 없습니다.

중첩 if, while문 가능합니다.

(test3)

***[A]가 100과 같을 때 까지 반복합니다***

[A]가 100이면 반복을 종료하는 반복문입니다.

“반복문의 끝” 상수와 같이 나와야 합니다.

[A]가 100이 아니라면 무한정 반복하게 됩니다.

***[A]가 100보다 작거나 같으면 반복합니다***

위의 코드와 반대 조건문으로 작동합니다.

[A]가 100보다 작아야만 계속 반복하게 됩니다.

“까지”라는 문장 성분이 나오지 않으면 조건문이 있는 지 판단하고 조건문에 따라 반복을 수행하게 됩니다.

(test4)

***[A]를 [A]에 1을 더한 값으로 설정합니다***

여타 언어에서 i = i + 1과 같습니다.

“[A]에 1을 더한 값”을 [A]에 넣게 됩니다.

여기에서부터 대명사의 개념이 나오게 됩니다.

것, 값, 나머지와 같은 대명사가 오면 앞의 서술어가 이를 수식해줄 수 있습니다.

즉 최종적으로 대명사가 서술어의 결과값으로 치환됩니다.

(test6)

***[A]에 1 더합니다***

조사의 일부를 생략할 수 있습니다. (다 생략도 가능하기는 합니다)

위처럼 조사를 생략하게 되면 최종적으로 서술어는

“1 더합니다”와 같이 인식하게 됩니다. 이 상태에서 서술어를 제외한 “1”을 분류 없는 문장 성분으로 지정합니다.

이후 서술어에서 문장 성분이 부족하다 판단 될 경우 분류 없는 문장 성분을 가져다 쓰게 됩니다.

응용하면, 위에서 “[A]를 [A]에 1을 더한 값으로 설정합니다” -> “[A]를 [A]에 1 더한 값으로 설정합니다” 처럼 쓸 수 있습니다.

***[A]를 3을 2로 나눈 나머지로 설정합니다***

대명사에서 특정한 서술어가 수식하느냐에 따라 서술어의 기능을 바꿀 수 있는 기능이 들어가 있습니다.

“나머지”라는 대명사는 다른 서술어가 올 때는 “값”, “것”과 같이 평범한 대명사처럼 작동하지만

“나누다”라는 서술어가 올 때만 “나누다”의 원래 기능인 나눗셈 대신 나머지 연산을 수행하게 됩니다.

즉 [A]에는 1.5가 아니라 1이 들어갑니다.

(test13)

***[A]가 3과 같고 [A]가 3보다 크다면***

여타 언어에서 A>3 && A == 3과 같은 구문입니다. (즉 절대 true가 안되는 구문)

“같고” 라는 접속조사가 쓰였는데, 서술어가 “이고”, “고”로 끝나면 AND로 인식하게 됩니다.

다르게 “거나”라는 접속조사가 쓰이면 OR로 인식하게 됩니다.

(test19)

***[A]를 123123의 자리수로 초기화합니다***

위에서 “의” 문법과 비슷한 맥락입니다.

숫자도 객체로 취급되므로 “의”를 통해 다양한 변환을 시켜줄 수 있습니다.

123123의 자릿수로 적으면 실제 123123의 자릿수를 반환하여 6을 반환받습니다.

***[A]를 123123의 각 자릿수의 합계로 설정합니다***

숫자 자료형도 있지만 배열도 있습니다.

자릿수 대신 각 자릿수를 가져오면 1,2,3,1,2,3으로 나뉘어진 배열을 반환받습니다.

그리고 배열에서 합계를 가져오면 1,2,3,1,2,3의 합계를 반환받게 됩니다.

위에서는 합계, 자릿수에 대한 예만 들었지만 언제든지 객체 클래스에서 수정,추가가 가능합니다.

(test20)

***1과 2로 xy좌표를 만듭니다***

“xy좌표”라는 이름을 가진 객체를 만듭니다(Vector2d).

xy좌표는 Vector2d를 구현하고 사전에 미리 추가해두었습니다.

1과 2는 각각 x,y로 매핑 됩니다.

나중에 해당 xy 좌표를 변수에 저장하고 [변수]의 x좌표와 같은 문장을 통해 x,y 값을 가져오거나 설정할 수 있습니다.

***[A]를 1과 2로 xy좌표를 만든 것으로 초기화합니다***

[A] 변수에 “xy좌표” 객체를 집어넣었습니다.

***[A]의 x좌표를 3으로 설정합니다***

***[A]의 x를 3으로 설정합니다***

[A]의 x좌표를 3으로 설정하게 됩니다.

(test23)

***[A]에 1을 더해 [A]에 저장합니다***

맥락상 이해할 수 있는 구문을 한번 구현해 봤습니다.

서술어엔 역시 목적어가 부족한 상태입니다. 부족한 목적어를 앞의 서술어의 값으로 채우게 됩니다.

(testFromFile)

testfile.txt에는 다음과 같은 구문이 적혀있습니다.

1부터 1000 사이의 소수를 구하는 구문입니다.

"1부터 1000까지 소수"를 출력합니다  
[i]를 2로 설정합니다  
[i]가 1000보다 작거나 같으면 반복합니다  
 [j]를 2로 설정합니다  
 [verify]를 true로 설정합니다  
 [j]가 [i]보다 작으면 반복합니다  
 [i]를 [j]로 나눈 나머지가 0과 같다면  
 [verify]를 false로 설정합니다  
 조건문의 끝  
 [j]에 1을 더해 [j]에 저장합니다  
 반복문의 끝  
 [verify]가 true와 같다면  
 [i]를 출력합니다  
 조건문의 끝  
 [i]를 [i]에 1을 더한 것으로 설정합니다  
반복문의 끝  
"소수 끝"을 출력합니다

이를 실행하면 1부터 1000 사이의 소수 리스트를 얻을 수 있습니다.

2, 3, 5, 7…