Parser Demo

Демонстрация и объяснение работы модуля парсеров в sudu-editor

Для чего нам парсинг?

- Подсветка синтаксиса
- Поиск локальных declaration/usages
- Построение интервального дерева

Процесс парсинга

Для построения лексических и синтаксических анализаторов используется парсер-генератор ANTLR

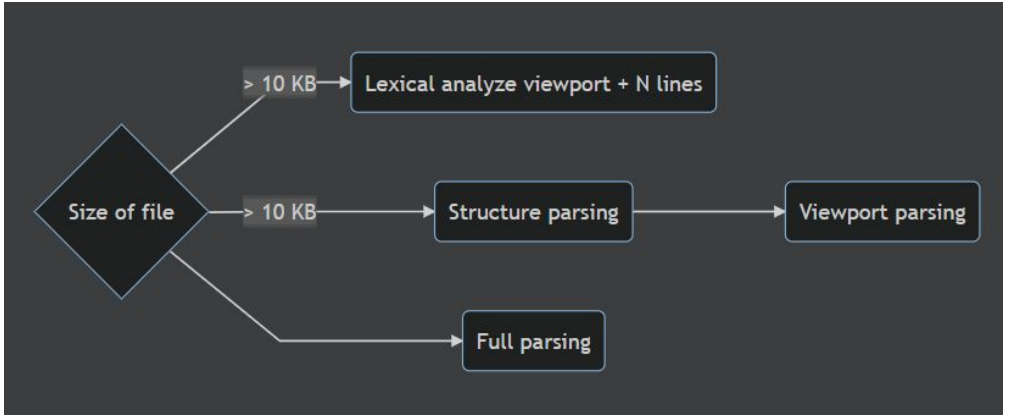
Грамматики для языков берутся из официального github-репозитория ANTLR

Сама библиотека содержит некоторые конфликтующие с TeaVM вещи (WeakHashMap, CopyOnWriteArrayList, System.err.format), поэтому используем совместимую с TeaVM версию ANTLR

Типы парсинга

Если файл большой, парсинг может занять продолжительное время, а какой-то результат пользователь хочет увидеть сразу. Поэтому парсинг файла разделен на следующие этапы:

1. Лексический анализ первых N линий
2. Парсинг структуры файла + вьюпорта пользователя
3. Полный парсинг



Процесс парсинга. Передача результата

Результат работы любого из анализаторов передается затем в ParserUtils в виде массива int[] result, где позже из него собирается Document. Подобный способ передачи результата вызван особенностями многопоточности в Web-браузере.

В result хранится информация о:

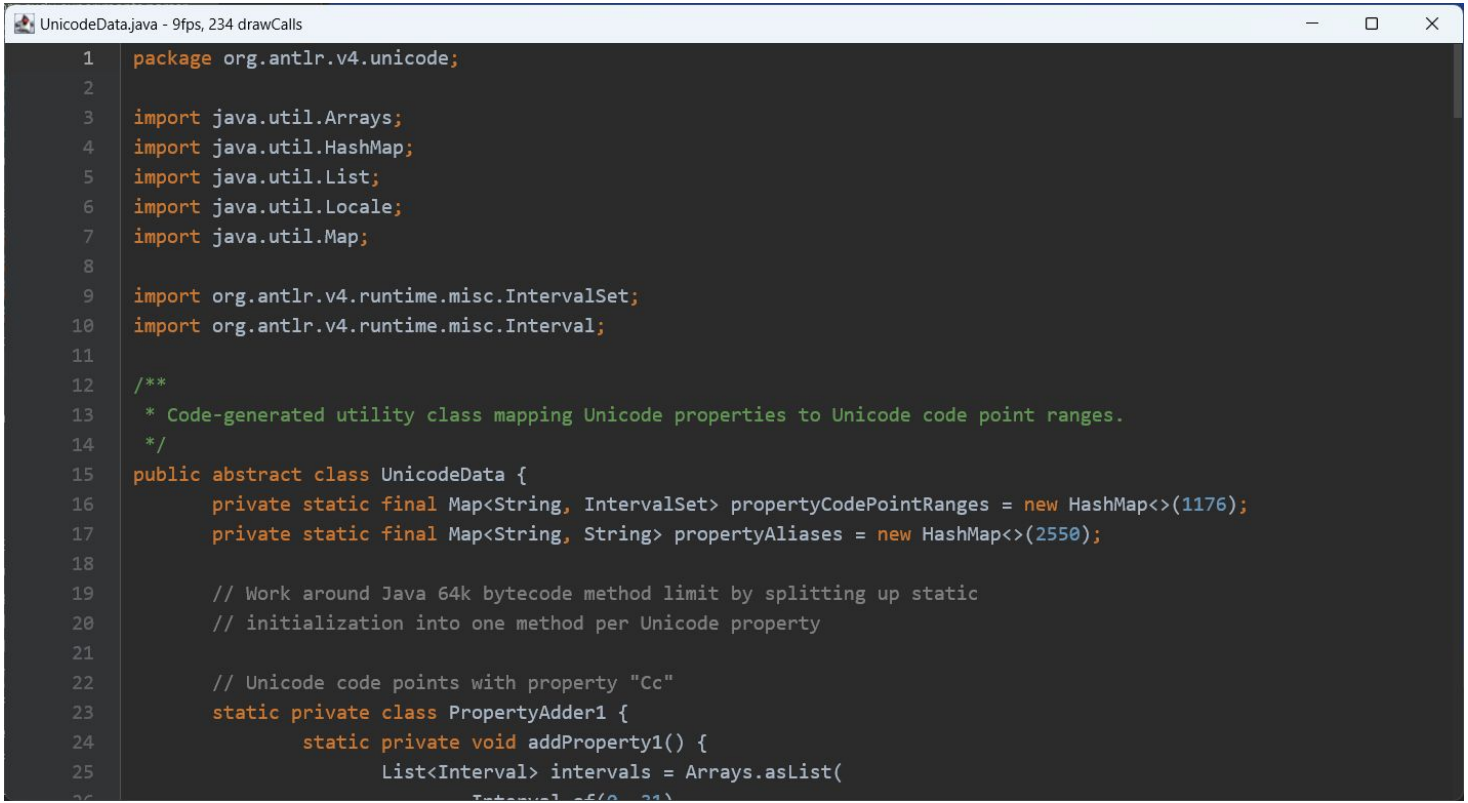
1. Количестве линий документа
2. Для каждой линии - кол-во находящихся в нем элементов кода
3. Для каждого элемента кода - индексы начала и конца в изначальном тексте, тип и стиль элемента
4. Количестве элементов для каждой линии
5. Структуре файла в виде дерева интервалов
6. Найденные пары declaration-usage

Процесс парсинга. Лекс. анализ

Лексический анализ в отличие от синтаксического можно выполнить достаточно быстро. Поэтому, при открытии файла мы можем произвести лексический анализ строк, попадающих во вьюпорт пользователя и еще N строк из его окрестности и, получив последовательность токенов, быстро подсветить некоторые ключевые слова и литералы.

При таком парсинге нельзя сделать никаких представлений о структуре документа

Процесс парсинга. Лекс. анализ

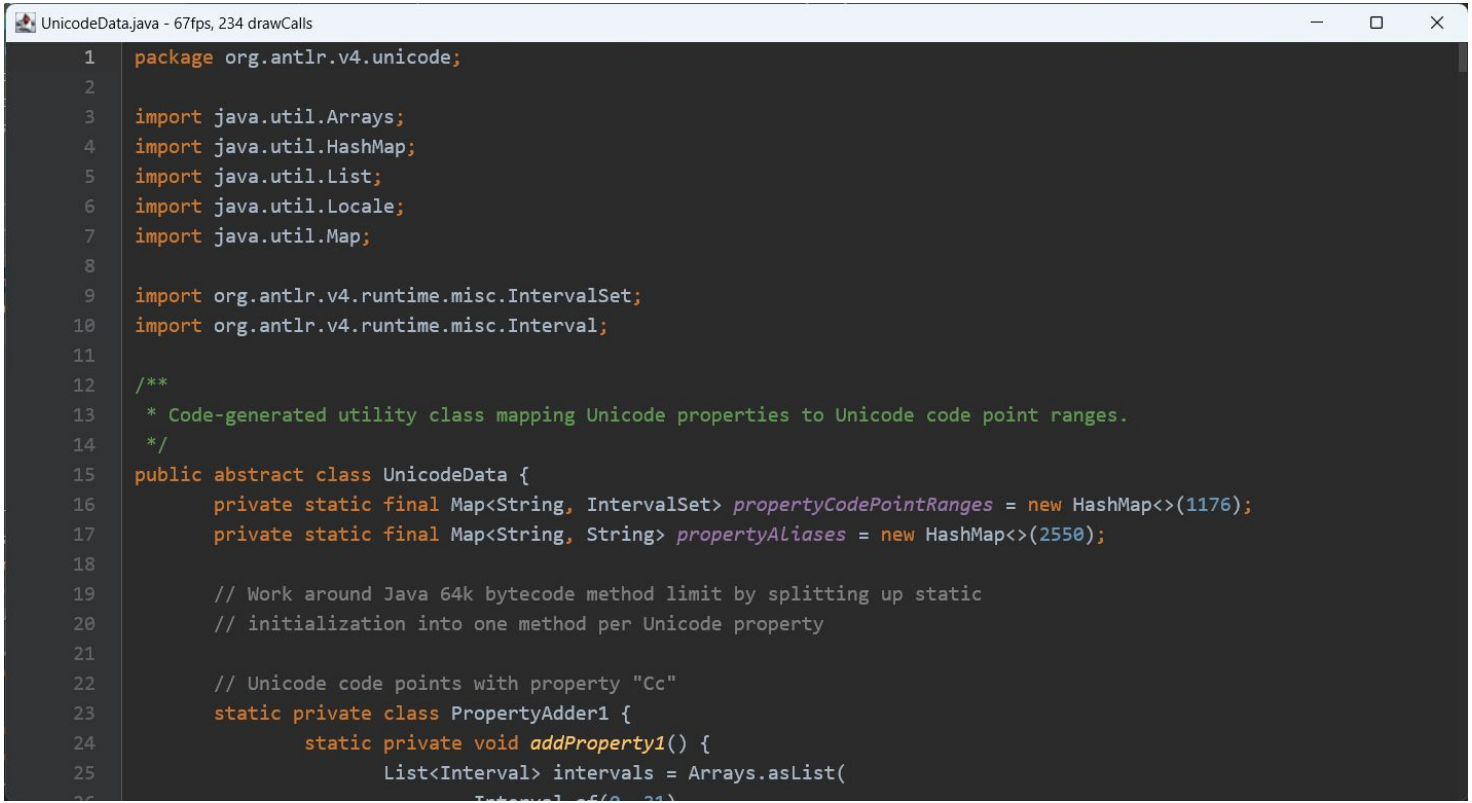


Процесс парсинга. Структурный анализ

Ключевая идея – проведение синтаксического анализа полного исходного файла, но по сильно упрощенной грамматике, которая ориентируется только на некоторые ключевые конструкции языка, а остальные символы – пропускает. Например в языке Java – некоторые ключевые слова и символы, а также скобочные последовательности могут дать нам представление о том, какой перед нами элемент структуры. Все же остальные токены – игнорируем. На выходе получаем дерево интервалов.

Сразу после этого парсинга, мы можем запустить полный парсинг, но только тех интервалов, которые в настоящий момент находятся на вьюпорте пользователя, а это заметно меньше чем полный файл.

Процесс парсинга. Структурный анализ



Процесс парсинга. Сравнение типов парсинга

Имя	Размер	250 линий + Viewport		Структура + Viewport		Полный Парсинг	
UnicodeData.java	4417 Кб	14 ms	94 ms	113 ms	1181 ms	11767 ms	63488 ms
EUC_TW_OLD.java	2302 Кб	7 ms	41 ms	51 ms	679 ms	1592 ms	9025 ms
TestBigObj.java	1603 Кб	6 ms	36 ms	187 ms	2175 ms	445 ms	2764 ms
INDIFY_Test.java	945 Кб	6 ms	17 ms	90 ms	885 ms	1780 ms	7714 ms
JavaParser.java	428 Кб	5 ms	11 ms	27 ms	350 ms	961 ms	5074 ms

Desktop + **Web**: Windows 10, 16 Гб RAM, Intel Core i7-9750H 2.6 GHz, SSD

Процесс парсинга. Обход деревьев

В процессе обхода правила грамматики обрабатываются таким образом, чтобы запоминать местонахождение и типы полей, методов, конструкторов, аргументов функции, локальных переменных и т.п.

Затем при заходе в обращение к полю/переменной или вызов метода/конструктора ищем его объявление среди найденных

Инкрементальный парсинг

При редактировании, структура файла может измениться и тогда понадобится обновить подсветку текущего сегмента. Однако, заново запускать полный парсинг на всем файле займет слишком много времени. Поэтому при внесении в документ изменений, также вносятся изменения и в дерево структуры. При этом соответствующий узел дерева помечается как нуждающийся в обновлении. После того как пользователь закончил печатать, к интервалам, нуждающимся в обновлении, применятся полный парсинг. Это позволяет поддерживать корректную структуру документа и правильную подсветку синтаксиса.

Известные проблемы. Несоответствие файла грамматике языка

Если файл не соответствует грамматике языка, то выполнить корректный синт. анализ не получится. В таком случае будут нужным образом будут подсвечены лишь ключевые слова, литералы и т.п.

Пример: в процессе препроцессинга в C++, макросы заменяются их фактическими значениями. До препроцессинга, файл может быть синтаксически некорректен, а после - корректен

Известные проблемы. Несоответствие файла грамматике языка

main.cpp

```cpp
#include <iostream>
#include "macro.h"

MAIN
    std::cout << "Hello, World!" << std::endl;
    return 0;
}

```

macro.h

```cpp
#define MAIN int main() {

```

Известные проблемы. Инкрементальный парсинг

Как было сказано выше, при редактировании файла парсится только тот интервал, где было внесено изменение. Таким образом, если в интервале находится использование поля, которое было объявлено вне этого интервала, то его declaration не может быть найдено.

Также при инкрементальном парсинге становятся неактуальными хранящиеся позиции declaration/usages.

Известные проблемы. Невозможность найти declaration

В процессе анализа, нам могут повстречаться выражения чей declaration мы не можем найти.

Например, это могут быть выражения, не объявленные в анализируемом файле, а импортированные в него. А так как мы имеем доступ только к одному файлу, то и найти их не получится

Известные проблемы. Невозможность найти declaration

```
1  import static org.sudu.experiments.Import.NAME; // String
2
3  public class Main {
4
5    void resolve() {
6      fun(NAME);  // ???
7    }
8
9    public void fun(int a) {}
10
11   public void fun(boolean a) {}
12
13   public void fun(String a) {}
14
15 }
16
```

Ссылки

Репозиторий с презентацией

sudu-editor

TeaVM compatible версия ANTLR

Грамматики ANTLR4