

## Курсова Работа: Част Втора

### До вторник, 18. ноември 2025 г. 23:59

#### 1 Обзор на курсовата работа

Курсовата работа се състои от четири части, които съответстват на четирите основни фази на един компилатор—лексикален анализатор, синтактичен анализатор, семантичен анализатор и генератор на код. Тук пропускаме фазата за оптимизации, понеже целта ни е да създадем възможно най-простия работещ компилатор. Всяко задание води до създаването на работеща фаза на компилатора, която да може да взаимодейства с останалите фази. Ще реализирате вашият компилатор на C++.

Първата част от курсовата работа изискваше от вас да създадете *лексикален анализатор* за програмния език COOL. Втората (тази) част от курсовата работа изисква от вас да създадете *синтактичен анализатор* (също наричан *parser*) за COOL. Ще използвате софтуерния инструмент ANTLR4, за да генерирате своя синтактичен анализатор. Заданието е да създадете контекстно-свободна граматика описваща синтаксиса на COOL в подходящия формат. ANTLR4 ще генерира C++ кода, който ще разпознава синтактичната структура на дадена COOL програма според вашите правила. За да използвате генерирания код, ще трябва да модифицирате и управляващата програма (наричана също *driver*), така че да отпечатва произведеното абстрактно синтактично дърво на програмата. Предоставени са редица тестове демонстриращи очакваното поведение на вашия синтактичен анализатор. Също така са предоставени и инструменти, които улесняват компилацията и изпълнението на кода ви.

Следната документация ще ви бъде от полза по време на работа:

1. [Ръководството за COOL \(на английски\)](#), особено раздел 11 – Cool Syntax, основно на страница 16; както и
2. The Definitive ANTLR4 Reference от Terence Parr.

Разрешено е да работите индивидуално или по двойки. Ако работите по двойки, всеки е длъжен да предаде общата работа поотделно.

## 2 Процес на Работа

За да започнете работа, изтеглете последната версия на докер контейнера. Ако сте работили по първата част от курсовата работа и искате да преизползвате лексикалния си анализатор, копирайте .g4 файла си за него в работната папка. Обърнете внимание, че LexerDriver.cpp няма да ви е директно нужен, макар да може да е полезно да преизползвате част от него (например функцията за отпечатване на името на тип на жетон).

```
docker pull ghcr.io/aristotelis2002/uni-cool:cw2
<cd to your work dir>
docker run -it --name cc-cw2 -v "$(pwd):/home/student/my-code" \
    ghcr.io/aristotelis2002/uni-cool
```

Обърнете внимание на това, че трябва да ползвате етикет на контейнера cw2-arm, вместо cw2, ако сте с ARM процесор (по-нов macbook компютър).

Свалете шаблона за тази част от курсовата работа и го копирайте в папка, в която ще работите.

```
git clone https://github.com/Aristotelis2002/uni-coolc temp
cp -r temp/cw2template cw2 && rm -rf temp && chmod -R g+w cw2
cd cw2 && chmod +x ./tools/*.sh
```

Шаблонът съдържа следните неща:

1. папка src, която съдържа шаблон на граматиката и прост драйвер; папката съдържа също така и списък със жетоните от служебния lexer;
2. папка tools, която съдържа инструменти за компилиране и тестване; и
3. папка tests, която съдържа тестовите файлове за оценяването;
4. папка lib, която съдържа библиотека със служебния lexer; и
5. папка include, която съдържа header за служебния lexer.

Кодът, който трябва да промените за това задание се намира във файловете “CoolParser.g4” и “drivers/ParserDriver.cpp” в папката src.

След промяна по граматиката или драйвера, рекомпилирайте синтактичния анализатор чрез скрипта “tools/build.sh”. За да започнете, извикайте този скрипт преди да сте направили каквито и да е промени и потвърдете, че той успешно завършва.

```
student@b9404ee896a3:~/my-code/cw2$ ./tools/build.sh
-- The CXX compiler identification is GNU 13.3.0
-- Detecting CXX compiler ABI info
```

```
-- Detecting CXX compiler ABI info - done
[...]  
[100%] Built target parser
```

Шаблонното решение успешно минава един от тестовете. За да проверите това, изпълнете `tools/test-parser.sh` с аргумент `001`.

```
student@b9404ee896a3:~/my-code/cw2$ ./tools/test-parser.sh 001  
Test 001.class PASSED
```

Този скрипт е доста приказлив, така че е добавена възможност да изпълнява само някои от тестовете, подвайки префикс на името на теста, който да се изпълни. Така `./tools/test-parser.sh 01` ще изпълни 10 теста – `010`, `011`, `012`, и т.н. до `019`.

### 3 Задание

Задачата ви е да напишете ANTLR4 правила, които да дефинират синтактична спецификация на програмният език COOL. Тази спецификация е описана в [Ръководството за COOL \(на английски\)](#), особено раздел 11 – Cool Syntax, основно на страница 17. По-важна част от документацията е Фигура 1., която е на страница 16 и съдържа Бекус–Наур форма на COOL синтаксиса, която трябва да преведете до ANTLR4 (мета-)синтаксис.

За тази част от курсовата работа не е нужно да пишете специален C++ код директно в `.g4` файла. C++ кодът, който трябва да напишете, се намира в управляващата програма дефинирана в `drivers/ParserDriver.cpp`. Изискването е този код да отпечата абстрактното синтактично дърво произведено от синтактичния анализатор (класа `CoolParser`). Шаблонът съдържа достатъчно код, така че да можете да започнете итеративна работа.

Един начин да подходите към решението е да изпълнявате тестовете подред. Ако някой тест не минава да промените решението си и да опитате пак, докато тествания скрипт не даде резултат `PASSED`. За повече информация, вижте предния раздел: Раздел 2.

Преди да започнете работа, прочетете внимателно раздел 11 от ръководството.

Вашият синтактичен анализатор трябва да е стабилен—трябва да приема всеки възможен вход. Трябва да се погрижите в случай на грешка, програмата да завършва по безопасен начин. Не е приемливо да се стига до `crash`-ове или `segmentation fault`. Желателно е да отпечатвате всички възможни грешки, но оценяващата програма проверява само съвпадение в съобщението от първата грешка. Това е така, понеже най-полезното докладване на грешки е отчасти субективна величина. И все пак, ако смятате, че сте успели да докарате точно същите грешки като в тестовите файлове ще получите 20% бонус точки към оценката на тази част от курсовата работа.

**За да е възможно автоматичното оценяване, изходът от вашия синтактичен анализатор трябва точно да съвпада с предоставените тестове.** Вижте раздел [2](#) за повече информация как да компилирате и тествате решението си. Изключението е, че в тестовите очакващи грешка, оценяващия скрипт проверява само първия ред.

## 4 Общи Насоки

ANTLR4 е модерен инструмент за генерирането на лексикални и синтактични анализатори. По подразбиране, инструментът генерира комбинирана библиотека, която изпълнява и двете фази едновременно. Възможно е да се използва, обаче, и за генериране на фазите поотделно. В предната част от курсовата работа използвахте ANTLR4 за да генерирате само лексикален анализатор. В тази част ще го използвате, за да генерирате само синтактичен анализатор, като ще го свържете с лексикалния анализатор от предната част. Ако не сте предали лексикален анализатор за предната част или не е напълно завършен, може да ползвате служебния такъв, предоставен с материалите за курсовата работа.

ANTLR4 компилира вашия файл със синтактична граматика (например, “Parser.g4”) към C++ код. При изпълнение може да се избере дали да се генерира listener, или visitor, като ние ще ползваме второто. Чрез тази библиотека ще можете да дефинирате преминавания (passes) през абстрактното синтактично дърво на програмата. Пример за такова преминаване е отпечатващ инструмент, който ще ви е нужен за пълно решение на тази част от курсовата работа. За щастие, няма нужда да гледате или разбирате генерирания код, за да го ползвате.

### 4.1 Обща структура на входния файл

Синтактичната граматика се дефинира в ANTLR4 по следния начин:

```
parser grammar ИмеНаСинтактичнияАнализатор;  
options { tokenVocab=ИмеНаЛексикалнияАнализатор; }  
Граматика
```

В един смисъл, заданието е по-лесно от предното, понеже не е нужно да пишете допълнителен C++ код във файла с граматиката.

Частта ”граматика” е дефиниция на контекстно-свободна граматика съответстваща на синтактичната структура на езика. Дефиницията трябва да следва правилата, определени от спецификацията на ANTLR4. Поради тази причина е силно препоръчително да се сдобите с копие на “The Definitive ANTLR4 Reference” за изпълнението на тази част от курсовата

работа. Все пак, тук ще се опитаме да опишем най-важните аспекти от спецификацията, но е възможно те да не са достатъчни за пълно решение на заданието.

## 4.2 Нетерминали

В ANTLR4 имената на синтактичните правила (нетерминалите) започват с малка буква. След името следва двоеточие, след което списък от правила, разделени с вертикална черта. Списъкът от правила завършва с точка-и-запетая.

Всяко правило от списъка представлява регулярен израз, съдържащ нетерминали и терминали в произволна комбинация. Терминалите са жетоните от лексикалния анализатор. Лексемите може да се ползват и директно, стига типът на жетона да позволява само една лексема. Например, ако е дефиниран следния тип жетон в лексикалната спецификация, АТ: '@', в синтактичната спецификация може да се ползва директно '@'. Ако лексемата не съответства директно на тип жетон, ANTLR4 ще върне грешка “cannot create implicit token for string literal in non-combined grammar”.

Да разгледаме следния пример:

```
arith : NUM '+' NUM  
      | NUM '-' NUM;
```

Тук нетерминалът е arith и граматиката за него съдържа две правила: едно за сбор и едно за разлика. Отново, нужно е и трите вида жетон – NUM, плюс и минус – да са дефинирани в лексикалната спецификация, определена в началото на файла чрез tokenVocab=.

## 4.3 Приоритет и Асоциативност

Правилата написани по-рано във файла с граматиката са “с по-висок приоритет”. Възможно е това да доведе до объркване, понеже операции с по-висок приоритет всъщност се появяват по-късно в АСД, но това е правилно. За постигане на еднакъв приоритет за две операции, те трябва да са реализирани чрез общо правило (на един ред от граматиката).

Операциите са ляво-асоциативни по подразбиране. Възможно е да се специфицират като дясно-асоциативни, чрез добавяне на <assoc=right> след двоеточието или правата черта <sup>1</sup>. За постигане на “не-асоциативни” операции, трябва да се напише специално преминаване, което да засича грешната употреба и да отпечата грешка.

---

<sup>1</sup>тук документацията е грешна; синтаксисът е променен във версия 4.2: [тема в StackOverflow](#)

## 4.4 Лява Рекурсивност

Лявата рекурсивност е позволена, стига да е директна (един нетерминал може да се запише в последователност, започваща пак с него). Индиректната лява рекурсивност е забранена и води до отпечатване на грешка “The following sets of rules are mutually left-recursive ...”. Това ограничава гъвкавостта при писането на граматика и затова по-сложните проблеми трябва да се решават в специализирано преминаване през АСД след неговото построение (конкретно – реализацията на “не-асоциативни” оператори).

## 4.5 Първоначална Граматика

Шаблонният код от тази част включва следната първоначална граматика:

```
parser grammar CoolParser;  
options { tokenVocab=CoolLexer; }  
program: (class ';'')+ ;  
class   : CLASS TYPEID '{' '}' ;
```

Тя предполага, че лексикалният анализатор е бил дефиниран в CoolLexer.g4 и че съдържа дефиниции на видовете жетони CLASS, TYPEID, '{', и '}'. Няма да е нужно да промените нетерминалът program, но ще трябва да допълните дефиницията на class, както и да добавите нови нетерминали, които рекурсивно да изграждат цялата синтактична структура на програмния език COOL. Задачата не е лека, но крайният резултат е учудващо кратък и изразителен.

Шаблонният пакет също съдържа файлове за служебния лексикален анализатор:

1. src/CoolLexer.tokens,
2. include/CoolLexer.h и
3. lib/liblexer\_gen\_code.

Ако искате да ползвате лексикалния анализатор, който сте предали за част първа от курсовата работа, може да изтриете тези файлове от вашето копие на шаблонния код.

## 4.6 Visitor Pattern

В режим -visitor (този, който се използва от скрипта tools/build.sh, който сме ви предоставили) за всеки нетерминал, например name, ANTLR4 генерира функция visitName. Ако граматиката ви се казва Foo.g4, тогава се генерира и клас FooBaseVisitor, който съдържа

тези visit методи. Основната част от работата ви, освен дефиницията на правилна граматика, ще бъде да реализирате клас наследяващ този BaseVisitor. Шаблонният код съдържа първоначална дефиниция на този клас – TreePrinter.

Може да промените TreePrinter както пожелаете, стига промяната да доведе до работещ синтактичен анализатор. Основният модел на изпълнение е следният. Функцията print е входната точка към функционалността на класа. На свой ред, тя извиква visitProgram, с което се започва посещаването на всички елементи на АСД. Това посещаване се изпълнява автоматично, стига да не презапишете някоя от visit функциите. Когато го направите (ще е необходимо), е желателно да извикате отново друга visit функция, за да продължи посещаването надолу по дървото. Например, ако сте презаписали visitProgram до следното:

```
any visitProgram(CoolParser::ProgramContext *ctx) override {  
    cout << "_program" << endl;  
    visitChildren(ctx);  
    return any{};  
}
```

в случай че visitChildren липсваше, нямаше да се извика транзитивно visitClass. Така преминаването през програмата няма да е пълно. В някои случаи това е желателно, но при TreePrinter, който цели да отпечата цялата програма, това е бърк.

Всяка visit функция получава като аргумент обект от съответен Context клас. Чрез него могат да се достъпят елементите от дясната част на кое да е правило за нетерминала. За граматиката по-горе, ProgramContext съдържа class\_() метод за достъп, който в този случай ще върне вектор, понеже е възможно класовете да са повече от един (заради оператора + в правилото). Отделните класове могат да се достъпят или чрез итериране по този вектор, или чрез свързания метод class\_(index). ClassContext, пък, от своя страна, има методи CLASS(), TYPEID(), OPAREN() и CPAREN(). Ако дефиницията остане каквато е, в дясното правило за class няма друг нетерминал, така че не е нужно да се извиква друга visit функция, ако презаписвате visitClass.

## 4.7 std::any

Генерираният “посетител” на АСД използва std::any като тип на върнатата стойност от всеки visit метод. Това е механизъм, който позволява връщането на стойност от който и да е тип, стига тази стойност да бъде опакована в std::any, например така:

```
return std::any{std::string("hello")};
```

Обърнете внимание на допълнителното обвиване на “hello” в std::string. Няма да изпадаме в детайли, но за други стойности това не е нужно. Важното е да знаете, че в случай, че

искате да използвате символен низ като стойност, сме ви спестили с този пример много главоболия :-). За да използвате стойността в част от кода, която директно извиква някоя `visit` функция, трябва да ползвате `std::any_cast`, за да я разопаковате.

```
auto message = std::any_cast<std::string>(visitClass(ctx->class(0)));
```

За да обобщим, следното е друг пример:

```
std::any visitClass(CoolParser::ClassContext *ctx) override{
    return std::any{13};
}
...
auto unfortunate = std::any_cast<int>(visitClass(ctx->class(0)));
```

В случай, че се опитате да разопаковате стойност в тип, който не ѝ съответства, `any_cast` ще хвърли изключение от тип `std::bad_any_cast`. Така че, може да получите следната грешка по време на изпълнение:

```
terminate called after throwing an instance of 'std::bad_any_cast'
```

## 4.8 CoolLexer.h

Служебният лексикален анализатор предоставя две функции, които може да ви се наложи да ползвате. Едната е `get_csl_text`, а другата – `get_bool_value`. Коментарите в `CoolLexer.h` би трябвало да са достатъчни, но за пълнота, ще ги добавим и тук.

Методът `get_csl_text` приема за аргумент номер на символа от входния низ, където жетонът от тип `STR_CONST` започва. Върнатата стойност е от тип `std::string`. Следва примерна употреба на метода:

```
int char_index = ctx->STR_CONST()->getSymbol()->getStartIndex();
print_escaped_string(cout, lexer->get_csl_text(char_index));
```

За целта, кодът, който има нужда да използва метода, трябва да има достъп до указател към обекта от тип `CoolLexer`, който е използван за лексикален анализ. В тази извадка от програма, `ctx` е `Context` обект (виж Подраздел 4.6) на нетерминал, който някъде в дясната част на правилото си има терминал от тип `STR_CONST`.

Методът `get_bool_value` приема за аргумент номер на символа от входния низ, където жетонът от тип `BOOL_CONST` започва. Върнатата стойност е от тип `bool`. Следва примерна употреба на метода:

```
int char_index = ctx->BOOL_CONST()->getSymbol()->getStartIndex();  
cout << lexer->get_bool_value(char_index) << endl;
```

За целта, кодът, който има нужда да използва метода, трябва да има достъп до указател към обекта от тип CoolLexer, който е използван за лексикален анализ. В тази извадка от програма, ctx е Context обект (виж Подраздел 4.6) на нетерминал, който някъде в дясната част на правилото си има терминал от тип BOOL\_CONST.

## 5 Предаване на Решение

За да предадете решение, изпълнете “tools/grade-parser.sh” в контейнера. Този скрипт ще отпечата код за предаване, например “4:a87ff679a2f3e71d9181a67b7542122c”. Копирайте този код в специалния [формуляр за предаване в Мудъл](#). Освен това, трябва да качите и променения от вас код (само “CoolParser.g4” и “ParserDriver.cpp”) в специалната [папка за предаване в Мудъл](#). Ако сте променили C++ проекта, така че да включва и други файлове, качете и тях в папката.