Курсова Работа: Част Първа
До вторник, 4. 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 изисква от вас да създадете *лексикален анализатор* (също наричан *lexer* или *scanner*) за програмния език COOL. Ще използвате софтуерния инструмент ANTLR4, за да генерирате своя лексикален анализатор. Заданието е да опишете множество от жетони на COOL в подходящия формат. ANTLR4 ще генерира C++ кода, който ще разпознава вашите жетони. За да използвате генерирания код, ще трябва да модифицирате и управляващата програма (наричана също *driver*).

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

1. Ръководството за COOL (на английски), особено раздел 10 – Lexical Structure, основно на страница 15; 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 Въведение в работата с ANTLR4

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. Ще го използвате в този режим, за да генерирате само лексикален анализатор.

ANTLR4 компилира вашия файл с лексикална граматика (например, “Lexer.g4”) към C++ код, реализиращ краен автомат[1]. Този краен автомат разпознава регулярните изрази,

описани от вас във файла с лексикалната граматика. За щастие, няма нужда да гледате или разбирате генерирания код[2], за да го ползвате.

2.1 Обща структура на входния файл

Лексикалната граматика се дефинира в ANTLR4 по следния начин:

```
lexer grammar ИмеНаЛексикалнияАнализатор;
@lexer::members {
    Допълнителен C++ код, който да се копира в генерирания клас.
}
Видове жетони
```

Допълнителният C++ код е по избор. Позволява ви, да добавите функционалност директно в генерирания C++ клас, който ще представлява лексикалния анализатор по-късно в програмата.

Най-важната част от вашия лексически анализатор е частта, описваща видовете жетони. В ANTLR4 имената на лексическите правила започват с главни букви и трябва да са уникални. След името следва двоеточие след което регулярен израз, дефиниращ правилото. Всяко правило завършва с точка-и-запетая. Следва примерно правило:

```
NUM : [0-9]+ ;  // Правило дефиниращо числа
```

Важно уточнение е, че ако в даден момент повече от едно правило може да бъде приложено, ANTLR4 избира това, което разпознава най-дългата лексема (maximal munch). Например, ако дефинирате две правила:

```
A : [0-9]+ ;
B : [0-9a-z]+ ;
```

и анализаторът стигне до низ “2a”, тогава правило B ще се приложи, понеже то разпознава по-дълга лексема. Ако повече от едно правило разпознават една и съща лексема, тогава първото правило записано в “.g4” файла ще се приложи (first-come—first-served).

2.2 Команди

В случай, че искате да добавите допълнително поведение към разпознаването на лексема, може да добавите и *команда* към лексикалното правило. Всички седем команди можете да намерите в ръководството за ANTLR4, търсейки “Lexer Commands”. Синтаксисът използва стрелка надясно и позволява изброяването на няколко команди, разделени със запетая. Следва пример, в който добавяме командата ”skip”към правилото:

```
WS : [ \n]+ -> skip ;
```

2.3 Вградени действия

Традиционно, генераторите на лексикални анализатори широко използват *вградени действия* (embedded actions) за да надхвърлят ограниченията на регулярните езици. ANTLR3 също работи основно на този принцип. Подобрение е, че в ANTLR4 това не е задължително, за да се генерира работещ анализатор. Така могат да се генерира код на различни програмни езици, без да има нужда от промяна в граматиката.

И все пак, за по-сложни задачи (като лексикален анализатор на COOL, например) е неизбежно използването на вградени действия. Те се дефинират във фигурни скоби след регулярния израз и преди командата, ако има такава. Следва пример, в който се отпечатва текст, всеки път, когато се разпознае жетон от тип FRUIT:

```
@lexer::header {
#include <iostream>
}

FRUIT: ('apple' | 'orange') { std::cout << "tasty!\n"; } ;
```

Внимавайте да не пропуснете кръглите скоби, понеже тогава правилото ще се изпълни само за последния регулярен израз от обединението (в случая — 'orange')!

2.4 Режими

Докато пишете правилата на граматиката, може да ви се наложи да променяте приложимите правила, взависомост от жетони срещнати по-рано. Един начин да подходите към този проблем е, да използвате допълнителен C++ код, който да следи за състоянието на лексикалния анализатор. ANTLR4 също така предлага механизъм за смяна на *режима*. Достъпът до този механизъм е възможен чрез командите `mode(MODE_NAME)`, `pushMode(MODE_NAME)`, и `popMode`.

Правилата за даден режим се описват след декларация от вида `mode MODE_NAME;`. Правилата дефинирани преди първата декларация от този вид се считат към режим `DEFAULT_MODE`. Повече може да научите, като потърсите "lexical modes" в ръководството за ANTLR4. Следва пример, взет от там:

```
lexer grammar ModeTagsLexer;

// Default mode rules (the SEA)
```

```
OPEN  : '<'     -> mode(ISLAND) ;       // switch to ISLAND mode
TEXT  : ~'<'+ ;                         // clump all text together

mode ISLAND;
CLOSE : '>'     -> mode(DEFAULT_MODE) ; // back to SEA mode
SLASH : '/' ;
ID    : [a-zA-Z]+ ;                     // match/send ID in tag to parser
```

3 Процес на Работа

За да започнете работа, изтеглете последната версия на докер контейнера:

```
docker pull ghcr.io/aristotelis2002/uni-cool
<create a dir to work from; chmod g+w it; cd to it>
docker run -it --name cc-cw1 -v "$(pwd):/home/student/my-code" \
  ghcr.io/aristotelis2002/uni-cool
```

Промяната е, че в него вече има папка “/cw1template”, която трябва да копирате в работната си папка. След влизане в контейнера, изпълнете следното:

```
cp -r /cw1template cw1 && cd cw1
```

Шаблонът съдържа следните неща:

1. папка src, която съдържа шаблон на граматиката и прост драйвер;
2. папка tools, която съдържа инструменти за компилиране и тестване; и
3. папка tests, която съдържа тестовите файлове за оценяването.

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

След промяна по граматиката или драйвера, рекомпилирайте лексикалния 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@f1a1b3d92c74:~/my-code/cw1$ ./tools/build.sh
-- The CXX compiler identification is GNU 13.3.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
```

```
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found Java: /usr/bin/java (found version "17.0.16") found components: Runtime
-- Configuring done (0.3s)
-- Generating done (0.0s)
-- Build files have been written to: /home/student/my-code/cw1/build
[ 16%] Generating Cool lexer sources
[ 16%] Built target gen_lexer
[ 33%] Building CXX object CMakeFiles/generated_lexer.dir/home/student/my-code[...]
[ 50%] Linking CXX static library libgenerated_lexer.a
[ 66%] Built target generated_lexer
[ 83%] Building CXX object CMakeFiles/lexer.dir/home/student/my-code/cw1/src/d[...]
[100%] Linking CXX executable lexer
[100%] Built target lexer
student@f1a1b3d92c74:~/my-code/cw1$
```

Шаблонното решение успешно минава един от седемдесетте теста. За да проверите каква оценка би ви донесла работата досега във всеки един момент, изпълнете “tools/grade-lexer.sh”.

```
student@f1a1b3d92c74:~/my-code/cw1$ ./tools/grade-lexer.sh
Test 002.simple FAILED
Test 025.strcom FAILED
Test 026.strcom FAILED
...
Test 099.keywords FAILED
Test 100.keywords FAILED

Result: 1 out of 78 tests
Your submission code is: 1:c4ca4238a0b923820dcc509a6f75849b
```

При вас би трябвало последното съобщение да е на български. За да предадете решението си на тази част от курсовата работа трябва да качите променения от вас код (само “CoolLexer.g4” и “LexerDriver.cpp”) в специалната папка за предаване в Мудъл. Освен това, трябва да предадете и този код в специалния формуляр за целта в Мудъл. Ако работите по двойки, всеки трябва да предаде поотделно, за да получи оценка. Ако не предадете файловете или кода навреме (преди вторник, 4. ноември 2025 г. 23:59), ще получите 0 точки на тази част от курсовата работа.

За да разчистите работната си папка, използвайте скрипта “tools/clean.sh”.

4 Задание

Задачата ви е, да напишете ANTLR4 правила, които да дефинират лексикална спецификация на програмният език COOL. Тази спецификация е описана в Ръководството за COOL (на английски), особено раздел 10 – Lexical Structure, основно на страница 15. Вашите правила ще трябва да извършват определени действия за да:

1. запишат правилната лексема;
2. запишат правилния тип на жетона; и
3. запишат грешка, в случай, че такава възникне.

Преди да започнете работа, прочетете внимателно раздел 10 от ръководството. Трябва да декларирате поне следните типове жетони:

```
ASSIGN BOOL_CONST CASE CLASS DARROW ELSE ESAC FI IF IN
INHERITS INT_CONST ISVOID LE LET LOOP NEW NOT OBJECTID OF
POOL STR_CONST THEN TYPEID WHILE
```

За всеки тип жетон, вашето решение трябва да изпълнява определени действия, за да реализирате успешно лексикалната спецификация. Например, ако разпознаете жетон от тип BOOL_CONST, решението ви трябва да запише стойността на булевата променлива. (Шаблонът вече прави това.) Ако срещнете текстов низ, трябва да запишете верните стойности за всеки символ, след обработка на екраниращите последователности.

Вашият лексикален анализатор трябва да е стабилен—трябва да приема всеки възможен вход. Например, трябва да се справяте с грешки като EOF по средата на текстов низ или коментар. Друг пример са текстовите константи, които надвишават ограниченията. Ръководството на COOL съдържа всички грешки, които трябва да вземете предвид.

Трябва да се погрижите също така, в случай на грешка, програмата да завършва по безопасен начин. Не е приемливо да се стига до crash-ове или segmentation fault.

За да е възможно автоматичното оценяване, изходът от вашия лексикален анализатор трябва точно да съвпада с предоставените тестове. Вижте раздел 3 за повече информация как да компилирате и тествате решението си.

4.1 Обработване на Грешки

Всички грешки трябва да се предадат към синтактичния анализатор. Вашият лексикален анализатор не трябва да отпечатва нищо. Грешките се предават към синтактичния анализатор (втората фаза от компилатора) чрез специален вид жетон на име ERROR. На лице са следните изисквания за докладване на и възстановяване от лексически грешки:

- При срещането на невалиден символ (такъв, който не може да е част от лексема), самият символ трябва да се предаде като съобщението за грешка. Лексическият анализатор продължава от следващия символ.
- При докладването на номер на реда за низ, който обхваща повече от един ред, използвайте последния ред. В общия случай, номера на реда е там където жетона приключва. Това олеснява реализацията — в противен случай би ви се наложило да следите допълнителна информация.
- Когато един текстов низ е твърде дълъг, докладвайте грешката “String constant too long”. Ако текстовият низ съдържа невалидни символи (например нулевия символ), докладвайте това като “String contains null character”. Ако текстовият низ съдържа не екраниран нов ред, докладвайте това като “String contains unescaped new line”. Докладвайте само първата грешка за даден низ, след което продължете с анализа след края на низа. Края на низа е

  1. началото на следващия ред, ако низа съдържа не екраниран нов ред; или
  2. символът след затварящата в противен случай.

- Ако коментар съдържа край-на-файла (EOF), докладвайте тази грешка като ”EOF in comment”. Грешка е да жетонизирате съдържанието на коментара, само защото затварящия жетон липсва. Също и за текстови низове, ако край-на-файла се среща преди затварящите кавички, докладвайте тази грешка като ”EOF in string constant”.
- Ако срещнете ”*)”извън коментар, докладвайте тази грешка като ”Unmatched *) вместо да го жетонизирате като * и).
- Спомнете си от лекциите, че тази фаза на компилатора хваща само много ограничено множество от грешки. **Не проверявайте за грешки, които не са лексикални грешки в тази част от курсовата работа.** Например, не трябва да проверявате дали променливите са декларирани преди да се ползват. Убедете се, че напълно разбирате кои грешки се очаква лексикалният анализатор да хваща и кои — не, преди да започнете работа.

4.2 Текстови Низове

Вашият лексикален анализатор трябва да превърне екраниращите символи, които се срещат в текстовите низове в техните коректни стойности. Например, ако входната програма съдържа следните осем символа[1]:

```
" a b \ n c d "
```

[1]Символите са “двойни кавички”, “a”, “b”, “наклонена черта наляво”, т.н.; празните места са сложени за яснота в документа и не се броят.

... то вашият лексикален анализатор ще върне жетон от тип STR_CONST със семантична стойност от следните пет символа:

```
a b \n c d
```

където
n представлява символът за нов ред.

Следвайки спецификацията на страница 15 от ръководството за езика COOL, трябва да върнете грешка за текстов низ съдържащ нулевия символ. За сметка на това, последователността от два символа

```
\ 0
```

е позволена, но трябва да се преобразува в един символ: цифрата `0`, ASCII стойност 48.

5 Бележки по Реализацията

Управляващата програма (драйвера) изчита всички жетони наведнъж и ги отпечатва. Когато добавяте нови видове жетони в "CoolLexer.g4" ще трябва да добавяте и начина по който да се отпечатват в "driver/LexerDriver.cpp". В противен случай, драйвера ще отпечата "InvalidToken" и решението ви няма да се зачете за вярно. Лексемите състоящи се от единични символи се отпечатват с вид жетон самите себе си. (Вижте кода на драйвера и ще ви стане ясно.)

За жетоните, които представляват булеви променливи, стойността на променливата също се отпечатва. Трябва да направите същото и за текстовите низове и за жетоните, които представляват грешки. За целта, трябва да добавите C++ код в частта от граматиката, която се копира в генерирания клас (виж раздел 2).

При срещането на грешка, лексикалният анализатор трябва да генерира жетон с тип грешка. Може да използвате метода setType(ERROR) във вграденото действие или командата type(ERROR) в лексикалното правило. Вижте раздел 4.1 за съдържанието, което трябва да сложите в грешките. Подобно на булевите променливи и текстовите низове, трябва да добавите C++ код, който да позволява записването и четенето на грешките. Този код трябва да предоставя и интерфейс (метод на класа), чрез който драйвера да може да достъпи грешката за даден жетон от вид ERROR.

Срещането на символи в текстов низ, които не могат да бъдат отпечатани (ASCII стойност под 32) създават проблем за "LexerDriver.cpp". При отпечатване на стойността на `STR_CONST` те трябва да се отпечатат като шестнайсетична стойност с две цифри, например `<0x12>`. Също така, трябва да направите изключение за символите \n, \t, \b, и \f.

6 Предаване на Решение

За да предадете решение, изпълнете “tools/grade-lexer.sh” в контейнера. Този скрипт ще отпечата код за предаване, например “1:c4ca4238a0b923820dcc509a6f75849b”. Копирайте този код в специалния формуляр за предаване в Мудъл. Освен това, трябва да качите и променения от вас код (само “CoolLexer.g4” и “LexerDriver.cpp”) в специалната папка за предаване в Мудъл.

Бележки от Текста

[1]Всъщност, ANTLR4 генерира *Разширена Мрежа на Преходите, РМП* (Augmented Transition Network, ATN). РМП е структура базирана на крайните автомати. За тази част от курсовата работа може да смятаме РМП за еквивалентна на КА.

[2]И все пак за любопитните: генерираният C++ код просто кодира крайния автомат в двоичен формат. Помощната среда (runtime) на ANTLR4 след това интерпретира двоичния формат и разпознава лексемите, които той описва.