Tema 1
Analiză lexicală și sintactică

CPL

Cuprins

1 Obiective

Obiectivele temei sunt următoarele:

- Implementarea **specificațiilor lexicală și sintactică** ale limbajului Cool.
- Utilizarea instrumentului **ANTLR**.
- Întrebuințarea mecanismelor de **parcurgere** a arborilor de derivare (*listeners*/*visitors*).
- Construcția și parcurgerea **arborilor de sintaxă abstractă (AST)**.

2 Descriere

Tema abordează primele două etape ale construcției compilatorului pentru limbajul Cool, analiza lexicală și cea sintactică. Reprezentarea intermediară generată în aceste etape, în forma unui **AST**, va constitui punctul de plecare pentru

rezolvarea următoarelor două teme, de analiză semantică, respectiv de generare de cod.

În vederea testării, tema curentă va primi ca parametri în linia de comandă numele unuia sau mai multor fișiere conținând programe Cool, și va tipări la *standard output* **reprezentarea ierarhică** a programului, în formatul precizat în secțiunea 3.2, iar la *standard error*, eventualele **erori** apărute. În cadrul temei, veți genera **exclusiv erori lexicale**, în maniera precizată în secțiunea 3.1.2. Erorile sintactice vor fi cele generate automat de ANTLR.

Pentru exemplificarea funcționalității, pornim de la cele două programe Cool de mai jos:

```
-- p1.cl
class A {};

class B inherits A {};
```

```
-- p2.cl
class A {
    x : Int; #
};
```

Pentru acestea, obținem următoarele rezultate:

```
> java cool.compiler.Compiler p1.cl
program
  class
    A
  class
    B
    A

> java cool.compiler.Compiler p2.cl
"p2.cl", line 2:13, Lexical error: Invalid character: #
Compilation halted
```

Scheletul de pornire al temei gestionează următoarele aspecte:

- **Multiple** fișiere de intrare denumite în linia de comandă.
- Rularea **analizoarelor** lexical și sintactic definite de voi pe fiecare dintre fișierele de mai sus.
- Construcția unui **arbore global de derivare**, care agregă arborii de derivare individuali din toate fișierele prelucrate.
- Afișarea sugestivă a **erorilor**, cu informații despre fișierul în care a apărut eroarea, linia și coloana, tipul erorii (lexicală sau sintactică) și mesajul propriu-zis.

3 Cerințe

În continuare, sunt detaliate cerințele aferente celor două etape, de analiză lexicală, respectiv sintactică.

3.1 Analiza lexicală

Redactați în ANTLR specificația lexicală a limbajului Cool, pornind de la secțiunile 10 și 11 ale manualului acestuia.

Mai jos, sunt precizate anumite transformări ce vor fi aplicate asupra literalilor șir de caractere, precum și erorile lexicale ce trebuie generate.

3.1.1 Șiruri de caractere

În vederea obținerii reprezentării interne a literalilor șir de caractere întâlnite în programele Cool, se vor aplica transformările de mai jos. Puteți utiliza metoda `setText(String)` a clasei `Lexer` pentru a modifica dinamic lexemul aferent *token*-ului aflat în construcție. Ea poate fi invocată într-un bloc `{}`.

- Cele două caractere **ghilimele** de la începutul, respectiv sfârșitul unui literal, sunt eliminate din reprezentarea internă a acestuia.
- Secvența de **două caractere** `"\n"` (`\` și `n`) din cadrul unui literal, este reprezentată intern în forma unicului caracter `'\n'` (linie nouă). Similar pentru secvențele `"\t"`, `"\b"` și `"\f"` (vezi secțiunea 10.2 a manualului).
- Orice altă secvență `"\c"`, este interpretată drept caracterul `c`.

De exemplu, literalul `"12\n\a34"`, având lungime 10 (ghilimelele și caracterele *backslash* fac parte din lexem) va fi transformat în reprezentarea `12\na34`, având lungime 6, în urma eliminării ghilimelelor, a înlocuirii secvenței `"\n"` cu caracterul de linie nouă, subliniat, și a înlocuirii secvenței `"\a"` cu caracterul `'a'`.

3.1.2 Erori lexicale

Abordarea consacrată a erorilor lexicale este de a nu le genera direct din analizorul lexical, ci de a construi un *token* având o **categorie lexicală dedicată**, `ERROR`, și de a include în lexem mesajul de eroare. Ulterior, analizorul sintactic va decide ce este de făcut cu acest *token*, eventual afișând mesajul de eroare aferent.

Puteți utiliza metoda `raiseError(String)`, definită în schelet în cadrul analizorului lexical, în secțiunea `@members`, care primește drept parametru mesajul de eroare, și care poate fi invocată într-un bloc `{}`. Aceasta fixează categoria lexicală a *token*-ului aflat în construcție la `ERROR` și lexemul, la mesajul de eroare.

Erorile lexicale ce trebuie semnalate sunt următoarele:

- `String constant too long`, în cazul în care un literal șir de caractere, în urma transformărilor menționate în secțiunea 3.1.1, conține mai mult de 1024 de caractere.
- `String contains null character`, în cazul în care literalul conține caracterul terminator de șir, `'\0'` (ASCII 0). Atenție, nu este vorba de secvența de două caractere `"\0"`, transformată în caracterul `'0'`, conform regulilor de mai sus. În ANTLR, vă puteți referi la caracterul terminator prin `'\u0000'`.

- `Unterminated string constant`, dacă literalul conține caracterul `'\n'` (linie nouă) *non-escaped* (vezi secțiunea 10.2 din manual). Analiza lexicală continuă **după** caracterul de linie nouă.
- `EOF in string constant`, în cazul în care sfârșitul de fișier survine în interiorul unui literal.
- `Unmatched *)`, dacă marcajul de sfârșit de comentariu multilinie apare în exteriorul unui bloc de comentariu.
- `EOF in comment`, în cazul în care sfârșitul de fișier survine în interiorul unui comentariu multilinie.
- `Invalid character: c`, dacă este întâlnit un caracter nepermis, `c`. Mesajul de eroare va conține explicit caracterul problematic.

Pentru depistarea cazurilor de mai sus, se recomandă definirea de **reguli și alternative dedicate** acestora. Informația de linie și coloană corespunde începutului de literal, comentariu sau caracter problematic — comportamentul implicit al ANTLR la construirea unui *token*.

ATENȚIE! Analizorul lexical va fi scris în așa fel încât toate cazurile de eroare să fie tratate prin intermediul unui *token* `ERROR`. Indiferent de corectitudinea programului de intrare, ANTLR trebuie să **NU** genereze eroare!

3.2 Analiza sintactică

Redactați în ANTLR specificația sintactică a limbajului Cool, pornind de la secțiunea 11 din manualului acestuia. **NU** veți genera explicit erori sintactice!

Pasul următor constă în parcurgerea arborelui de derivare generat de ANTLR și construirea **AST**. Acesta va fi la rândul său vizitat în vederea afișării sale. Unul dintre avantajele utilizării unui **AST** este posibilitatea **decuplării** reprezentării interne a programului de cea furnizată de arborele de derivare, puternic dependent de gramatica utilizată. Spre exemplu, deși sintaxa limbajului Cool permite atât *dispatch*-uri explicite (e.g. `obj.f()` sau `self.f()`), cât și implicite (e.g. `f()`), ambele ar putea fi reprezentate de **același** tip de nod AST, adăugând manual în cazul celor implicite un nod **imaginar** reprezentând obiectul `self`.

În continuare, este descrisă **formatarea** reprezentării ierarhice a unui program Cool analizat, în funcție de fiecare construcție de limbaj. Forma este a unui arbore, în care copiii unui nod au o indentare cu **două spații** mai mare decât cea a părintelui (vezi exemplul de afișare din secțiunea 2). Parcurgeți **fișierele de test** pentru exemple concrete (vezi secțiunea 4). Pentru concizie, vom folosi **notația** `<etichetă părinte> : <copil 1> ... <copil n>`. Ordinea copiilor este cea din specificația sintactică a limbajului (secțiunea 11 din manual), dar anumiți copii irelevanți lipsesc (de exemplu, parantezele și acoladele).

Program `program : <clasă 1> ... <clasă n>`

Clasă `class : <nume> <clasă părinte>? <atribut/metodă 1> ... <atribut/metodă n>`

Atribut `attribute : <nume> <tip> <inițializare>?`

Metodă `method : <nume> <formal 1>? ... <formal n>? <tip> <corp>`

Variabilă `<nume>`

Literal `<literal>`

Operator binar `<operator> : <operand 1> <operand 2>`

Operator unar `<operator> : <operand>`

Atribuire `<- : <variabilă> <expresie>`

***Dispatch* explicit pe un obiect** `. : <obiect> <clasă static dispatch>? <metodă> <parametru 1>? ... <parametru n>?`

***Dispatch* fără obiect explicit** `implicit dispatch : <metodă> <parametru 1>? ... <parametru n>?`

Decizie `if : <condiție> <ramură then> <ramură else>`

Buclă `while : <condiție> <corp>`

Construcție `let` `let : <locală 1> ... <locală n> <corp>`

Definiție de variabilă locală `local : <nume> <tip> <inițializare>?`

Construcție `case` `case : <expresie> <ramură 1> ... <ramură n>`

Ramură `case` `case branch : <nume> <tip> <corp>`

Bloc `block : <expresie 1> ... <expresie n>`

4 Testare

Odată ce ați finalizat etapa de **analiză lexicală**, o puteți testa independent decomentând porțiunea din metoda `cool.compiler.Compiler.main`, care începe cu comentariul `// Test lexer only`. Aceasta afișează, pentru fiecare *token* recunoscut, lexemul și categoria asociate.

După implementarea etapei de **analiză sintactică**, puteți rula testele executând metoda `cool.tester.Tester1.main`, care afișează statistici despre fiecare test în parte, precum și scorul obținut, din 100 de puncte.

Testele se află în directorul `tests/tema1` din rădăcina proiectului. Fișierele `.cl` conțin programe Cool de analizat, iar cele `.ref`, ieșirea de referință a temei. Pentru fiecare test, sistemul de testare redirectează intrarea și eroarea standard ale compilatorului către un fișier `.out`, pe care îl compară apoi cu cel de referință.

Având în vedere că testele verifică **incremental** funcționalitatea analizorului sintactic, le puteți folosi pentru a vă ghida **dezvoltarea** temei!

5 Structura scheletului

În vederea unei mai bune structurări a implementării, **sursele** sunt distribuite în mai multe pachete, după cum urmează:

`cool.compiler` Modulul principal al aplicației

`cool.lexer` Analizorul lexical

`cool.parser` Analizorul sintactic

`cool.tester` Modulul de testare

În plus, **testele** se găsesc în directorul `tests/tema1` din rădăcina proiectului.

6 Precizări

Respectarea următoarelor precizări este necesară pentru buna funcționare a temei și a modulului de testare.

- Verificați că *jar*-ul de ANTLR de pe mașina voastră este accesibil în cadrul proiectului.
- Asigurați-vă că plugin-ul de ANTLR generează codul Java în directorul **`src`**, și nu în `target` sau `gen`. Pentru aceasta, consultați ghidul de configurare din Laboratorul 0. De asemenea, bifați ambele opțiuni de generare pentru *listeners* și *visitors*.
- Pentru gestiunea **terminatorului de linie** (`\n` sau `\r\n`) în specificația lexicală, se recomandă o abordare modulară, prin definirea unei reguli de tip `fragment`, care să surprindă diversele forme pe care terminatorul le poate lua, și utilizarea acesteia în celelalte reguli.
- În cadrul specificației lexicale, acțiunea **`-> skip`** se aplică pe o întreagă regulă, indiferent de numărul alternativelor. Dacă doriți să **reduceți** efectul acțiunii la o singură alternativă a regulii, apelați-o ca metodă într-un bloc Java intern, în forma `{ skip(); }`.
- După ce finalizați implementarea etapei de analiză lexicală, copiați fișierul **`CoolLexer.tokens`** din pachetul `cool.lexer` în pachetul `cool.parser`, pentru ca analizorul sintactic să aibă acces la categoriile lexicale.
- Regula sintactică principală se numește `program`.
- În cadrul nodurilor de **AST**, veți reține obiecte **`Token`**, care includ atât lexemul relevant nodului curent, cât și linia și coloana din fișier la care începe lexemul. Aceste informații vor fi importante pentru afișarea AST-ului în tema 1 și a erorilor semantice în tema 2. De asemenea, tot în pregătirea temei 2, este util să rețineți încă de acum în nodurile de AST obiectele **`ParserRuleContext`**, aferente arborelui de derivare generat de ANTLR. Acestea vor fi utile pentru afișarea numelui de fișier în care a apărut o anumită eroare semantică.
- Arhiva încărcată pe **`vmchecker`** va conține doar directorul `cool`.

7 Referințe

1. Manualul limbajului Cool.
   https://curs.upb.ro/2022/mod/resource/view.php?id=25343