Tema 2
Analiză semantică

CPL

Cuprins

1 Obiective

Obiectivele temei sunt următoarele:

- **Rezolvarea simbolurilor** din programele Cool, utilizând tabele de simboluri.
- **Verificarea tipurilor** definițiilor și expresiilor din programele Cool, prin implementarea regulilor de tipare din manualul limbajului.

2 Descriere

Tema abordează cea de-a treia etapă a construcției compilatorului pentru limbajul Cool, analiza semantică. Aceasta pornește de la reprezentarea intermediară generată în etapa anterioară, de analiză sintactică, în forma unui arbore de sintaxă abstractă (AST). Rezultatul acestei etape îl constituie o formă **adnotată** a reprezentării intermediare menționate mai sus, ce include informații despre **simbolurile** și **tipurile** din program.

Tema de față va primi ca parametri în linia de comandă numele unuia sau mai multor fișiere conținând programe Cool, și va tipări la *standard error* eventualele **erori semantice** apărute, sau nimic dacă programul este corect semantic. Programele de test vor fi **corecte lexical și sintactic**!

Mai jos, funcționalitatea este exemplificată pornind de la un program Cool simplu:

```
1 -- prog.cl
2 class A inherits B {};
3 class B inherits A {};
```

Pentru acesta, obținem următorul rezultat:

```
> java cool.compiler.Compiler prog.cl
"prog.cl", line 1:7, Semantic error: Inheritance cycle for class A
"prog.cl", line 2:7, Semantic error: Inheritance cycle for class B
Compilation halted
```

Scheletul de pornire al temei expune o funcție care asigură **afișarea** erorilor în formatul ilustrat mai sus.

3 Cerințe

Pentru început, definiți **clasele și metodele de bază** în domeniul global de vizibilitate, utilizând metoda `SymbolTable.defineBasicClasses()` din pachetul `cool.structures` (vezi secțiunea 5). Numele parametrilor formali ai metodelor de bază sunt cele din manualul limbajului Cool.

Așa cum a fost menționat în secțiunea 2, funcționalitatea de bază a temei constă în afișarea **erorilor semantice**, în cazul în care acestea există. În continuare, sunt descrise toate mesajele de eroare pe care este necesar să le generați, grupate pe construcții de limbaj. Utilizați în acest scop metoda `SymbolTable.error()` (vezi secțiunea 5). Pentru situații concrete, consultați testele (vezi secțiunea 4).

3.1 Definiții de clase

- `Class has illegal name SELF_TYPE`, dacă este definită o clasă cu numele `SELF_TYPE`.
- `Class <C> is redefined`, în cazul în care clasa este deja definită. Aici sunt incluse și cazurile de redefinire a claselor de bază, e.g. `Int`.
- `Class <C> has illegal parent <P>`, dacă părintele este `Int`, `String`, `Bool` sau `SELF_TYPE`.

- `Class <C> has undefined parent <P>`, dacă părintele nu este definit.
- `Inheritance cycle for class <C>`, în prezența unui ciclu de moștenire, de orice lungime.

3.2 Definiții de atribute

- `Class <C> has attribute with illegal name self`, întrucât atributele nu pot fi denumite `self`.
- `Class <C> redefines attribute <a>`, dacă atributul este deja definit în aceeași clasă.
- `Class <C> redefines inherited attribute <a>`, dacă atributul este deja definit într-o clasă strămoș.
- `Class <C> has attribute <a> with undefined type <T>`, dacă tipul atributului nu este definit.
- `Type <T1> of initialization expression of attribute <a> is incompatible with declared type <T2>`, dacă expresia de inițializare a atributului nu are un tip corect.

3.3 Definiții de metode

- `Class <C> redefines method <m>`, dacă metoda este deja definită în aceeași clasă. Cool **nu** permite supraîncărcarea metodelor cu tipuri diferite de parametri.
- `Class <C> has method <m> with undefined return type <T>`, dacă tipul întors nu este definit. De remarcat că numele `self` este permis pentru metode, datorită spațiilor de nume **diferite** utilizate de atribute și metode.
- `Class <C> overrides method <m> with different number of formal parameters`, dacă metoda este supradefinită cu alt număr de parametri.
- `Class <C> overrides method <m> but changes type of formal parameter <f> from <T1> to <T2>`, dacă metoda este supradefinită cu schimbarea tipului unui parametru. Numele `<f>` este aferent variantei supradefinite. Spre exemplu, dacă definiția originală este `f(x : Int)`, iar varianta supradefinită este `f(y : Bool)`, mesajul de eroare va reflecta numele `y`.
- `Class <C> overrides method <m> but changes return type from <T1> to <T2>`, dacă metoda este supradefinită cu alt tip întors.
- `Type <T1> of the body of method <m> is incompatible with declared return type <T2>`, în cazul în care corpul metodei nu are un tip corect.

3.4 Definiții de parametri formali

- `Method <m> of class <C> has formal parameter with illegal name self`, întrucât parametrii formali nu pot fi denumiți `self`.
- `Method <m> of class <C> redefines formal parameter <f>`, dacă parametrul este deja definit în cadrul aceleiași metode.
- `Method <m> of class <C> has formal parameter <f> with illegal type SELF_TYPE`, întrucât parametrii formali nu pot avea acest tip.
- `Method <m> of class <C> has formal parameter <f> with undefined type <T>`, dacă tipul parametrului nu este definit.

3.5 Construcția `let`

- `Let variable has illegal name self`, întrucât variabilele locale nu pot fi denumite `self`.
- `Let variable <l> has undefined type <T>`, dacă tipul variabilei nu este definit.
- `Type <T1> of initialization expression of identifier <i> is incompatible with declared type <T2>`, dacă expresia de inițializare nu are un tip corect.

3.6 Construcția `case`

- `Case variable has illegal name self`, întrucât variabilele locale nu pot fi denumite `self`.
- `Case variable <c> has illegal type SELF_TYPE`, întrucât este necesară precizarea unui tip concret.
- `Case variable <c> has undefined type <T>`, dacă tipul variabilei nu este definit.

3.7 Utilizări de variabile

- `Undefined identifier <i>`, dacă nu este accesibilă nicio definiție a variabilei în domeniul curent de vizibilitate.

3.8 Operatori aritmetici, relaționali și booleeni

- `Operand of <op> has type <T> instead of Int`, unde operatorul poate fi oricare dintre `+`, `-`, `*`, `/`, $\sim$, `<`, `<=`, dacă operandul are un alt tip decât `Int`.
- `Cannot compare <T1> with <T2>`, în cazul operatorului `=`, dacă cel puțin unul dintre cele două tipuri este `Int`, `String` sau `Bool`, iar celălalt este un alt tip (vezi manualul Cool).
- `Operand of not has type <T> instead of Bool`, dacă `not` este aplicat pe un operand cu un tip diferit de `Bool`.

3.9 Atribuiri

- `Cannot assign to self`, deoarece nu se pot realiza atribuiri către `self`.
- `Type <T1> of assigned expression is incompatible with declared type <T2> of identifier <i>`, în cazul în care expresia cu care se realizează atribuirea nu are un tip corect.

3.10 `new` și `isvoid`

- `new is used with undefined type <T>`, dacă tipul nu este definit.
- Nu există erori aferente lui `isvoid`, care întoarce întotdeauna `Bool`.

3.11 `while` și `if`

- `While condition has type <T> instead of Bool`, în cazul în care condiția nu are tipul corect.
- `If condition has type <T> instead of Bool`, în cazul în care condiția nu are tipul corect.

3.12 Apeluri de metode

- `Undefined method <m> in class <C>`, dacă metoda nu este definită. Pentru apeluri **dinamice**, de forma `e.f(e')`, metoda este căutată în clasa reprezentând tipul lui `e`. Pentru apeluri **statice**, de forma `e@C.f(e')`, metoda este căutată în clasa `C`.
- `Method <m> of class <C> is applied to wrong number of arguments`, dacă numărul parametrilor actuali nu corespunde.
- `In call to method <m> of class <C>, actual type <T1> of formal parameter <f> is incompatible with declared type <T2>`, dacă unul dintre parametrii actuali nu are un tip corect.
- `Type of static dispatch cannot be SELF_TYPE`, în cazul expresiei `e@SELF_TYPE.f(e')`.
- `Type <T> of static dispatch is undefined`, dacă se realizează un apel static de metodă pe baza unui tip nedefinit.
- `Type <T1> of static dispatch is not a superclass of type <T2>`, dacă, în expresia `e@T.f(e')`, `T` nu este o superclasă a tipului lui `e`.

3.13 Metoda `main`

- `No method main in class Main`, întrucât toate programele Cool impun existența unei clase `Main`, cu o metodă `main` (vezi manualul Cool).

4 Testare

Testele pot fi rulate executând metoda `cool.tester.Tester2.main`, care afișează statistici despre fiecare test în parte, incluzând erori nesemnalate sau semnalate în plus, precum și scorul obținut, din 100 de puncte.

Testele se află în directorul `tests/tema2` 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. Pentru a evita eroarea indusă de absența unei clase `Main` cu o metodă `main`, dar în același timp pentru a evita definirea acestei clase în fiecare fișier test, modulul de testare încarcă fișierul **`main.cl`** împreună cu fișierul curent de test.

Având în vedere că testele verifică **incremental** funcționalitatea analizorului semantic, le puteți folosi pentru a vă ghida **dezvoltarea** temei! Ele sunt structurate după cum urmează:

- Testele 1–17 **nu** necesită implementarea `SELF_TYPE` și a regulilor de tipare specifice. Acestea sunt utilizate propriu-zis doar în testele 18–20. Există mențiuni la `SELF_TYPE` și în primele 17 teste, dar **doar în poziții eronate**, pe care trebuie să le respingeți, e.g. definiție de clasă cu numele `SELF_TYPE`!
- Testele 1–5 abordează **definirea** corectă a entităților, presupunând rezolvarea referințelor la tipuri, dar **nu** și verificarea tipurilor pe baza regulilor de tipare.
- Testul 6 abordează rezolvarea **referințelor la variabile**, dar **nu** și la metode.
- Testele 7–17 abordează rezolvarea referințelor la variabile și **verificarea tipurilor** pentru construcțiile de limbaj, **cu excepția** apelurilor de metodă.
- Testele 18 și 19 abordează rezolvarea **referințelor la metode** și verificarea tipurilor în cazul **apelurilor de metodă**, introducând și **`SELF_TYPE`**.
- Testul 20 abordează **metodele de bază**, e.g. `abort()`.

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.structures` Găsiți aici metoda de **inițializare** a claselor și metodelor de bază, `SymbolTable.defineBasicClasses()`, precum și metoda de semnalare a **erorilor** semantice, `SymbolTable.error()`. Tot aici vă puteți defini și clasele pentru **simboluri**.

`cool.tester` Modulul de testare

În plus, **testele** se găsesc în directorul `tests/tema2` din rădăcina proiectului.

6 Precizări

- Având în vedere posibilitatea existenței de **utilizări anticipate** (*forward references*) pentru clase, atribute și metode, sunt necesare **multiple treceri** peste arborele sintactic.
- Pentru simplitate, testele presupun că, într-o primă etapă, este asigurată **corectitudinea ierarhiei de clase**, la nivelul definirii numelor și părinților claselor (vezi testul 1). Dacă apar erori la acest nivel, e.g. cicluri de moștenire, analiza semantică este **abandonată**, fără a trece mai departe la validarea definițiilor de atribute, metode etc.
- Rezolvarea simbolurilor și verificarea tipurilor **se împletesc** într-o manieră naturală. Spre exemplu, rezolvarea simbolului `x` este necesară pentru stabilirea tipului acestuia în expresia `x + 1`, în timp ce determinarea tipului subexpresiei `e` din expresia `e.f(x)` este necesară pentru rezolvarea simbolului `f`.
- Pentru **rezolvarea simbolurilor** în raport cu domeniile de vizibilitate corecte, puteți utiliza oricare dintre cele două variante discutate pe slide-urile 194–195 din curs.
- Pentru a înlesni generarea **cât mai multor erori semantice**, puteți preciza în anumite situații tipul unei expresii, **chiar dacă** subexpresiile conțin erori de tip. Spre exemplu, se poate considera că tipul expresiei `not 5` este `Bool`, chiar dacă operandul `5` are în mod evident tipul eronat `Int`, în loc de `Bool`.
- Pentru testele 1–17, care **nu** abordează `SELF_TYPE`, puteți considera că tipul lui **`self`** din clasa `C` este `C`, în loc de `SELF_TYPE(C)`.
- Având în vedere că erori diferite pot fi generate în treceri diferite, acestea nu vor avea o ordine fixă. Acest lucru nu este o problemă, deoarece modulul de testare **ignoră** ordinea generării erorilor.
- Metoda `SymbolTable.error()` primește drept parametri, pe lângă mesajul de eroare, **informație de context** din arborele de derivare generat de ANTLR (obiecte `Context` și `Token`), în vederea afișării fișierului, liniei și coloanei la care a apărut eroarea. Membrul `fileNames` din clasa `Compiler` conține asocieri între obiectele context aferente claselor Cool și numele fișierelor Cool în care aceste clase sunt definite. La primirea unui obiect context corespunzător unei erori, metoda `SymbolTable.error()` urmărește calea ascendentă către contextul strămoș aferent clasei Cool conținătoare, și consultă membrul `fileNames` pentru a determina numele fișierului asociat acestei clase. Prin urmare, veți valorifica obiectele `Context` și `Token` reținute în nodurile de AST în tema anterioară.

7 Referințe

1. Manualul limbajului Cool.
   https://curs.upb.ro/2023/mod/resource/view.php?id=18756