Trabalho Prático IV
Segunda, Junho 4, 2018

1 Introdução

Nesse trabalho, você irá implementar a semântica estática de Cool. Você irá usar a Árvore Sintática Abstrata (AST) construída pelo parser para verificar se o programa que está sendo compilado está em conformidade com a especificação da linguagem Cool. Seu analisador semântico deve rejeitar programas incorretos; para programas corretos, o componente deve coletar informações que auxiliarão na próxima fase. A saída do analisador semântico será uma AST anotada para ser usada na geração de código.

Você precisará referenciar as regras de tipagem, regras de escopo e outras restrições de Cool definidas no manual. Você também precisará adicionar métodos e estruturas de dados nas definições das classes da AST. As funções do pacote de árvores estão documentadas no *Tour of Cool Support Code*.

Em alto nível, seu analisador semântico terá que executar as seguintes tarefas:

1. Analisar todas as classes e construir um grafo de herança
2. Verificar se o grafo obtido é válido.
3. Para cada classe:
   (a) Atravessar a AST, coletando todas as declarações visíveis em uma tabela de símbolos.
   (b) Verificar se não há erros de tipagem.
   (c) Anotar a AST com os tipos

   Esta lista não é exaustiva. Você deve seguir a rigor o que está descrito no manual da linguagem.

2 Arquivos

Os arquivos deste TP se encontram na pasta assignments/PA4. Vocês podem modificar os seguintes arquivos:

- cool-tree.h: Nesse arquivo estão definidas várias classes associadas aos nós da AST. Vocês podem adicionar novas declarações a esse arquivo, mas não remova as que já estão lá.
- semant.cc: Esse é o arquivo principal para sua implementação da análise semântica. Ele possui alguns símbolos pré-definidos para sua conveniência e um começo de uma implementação da **ClassTable** para representar o grafo de herança. Você não é obrigado a usar ela

  O analisador semântico é invocado pela chamada do método **semant()** da classe **program_class**. A declaração de **program_class** está em cool-tree.h. Qualquer declaração de método que você adicionar em cool- tree.h deve ser implementado nesse arquivo.
- semant.h: Esse arquivo é o cabeçalho de semant.cc.

3 Caminhamento na árvore

Como um resultado do trabalho 3, seu parser constrói ASTs. O método **dump_with_types**, definido na maioria dos nós da AST, ilustra como atravessar a AST e coletar informações dela.

Sua tarefa de programação para esse trabalho é (1) atravessar a árvore, (2) coletar diversas informações sobre os nós, e (3) usar essas informações para garantir a semântica de Cool. Uma travessia na AST é chamamada de "passo". Você provavelmente precisará de ao menos dois passos na AST para verificar tudo.

Você provavelmente precisará anexar informações personalizadas nos nós da AST. Para fazer isso, você pode modificar diretamente cool-tree.h. As implementações dos métodos que você desejar adicionar devem estar dentro de semant.cc.

4 Herança

Os relacionamentos de herança especificam um grafo direcionado. Um requisito típico da maioria das linguagens com herança é que o grafo de herança seja acíclico. É função do seu analisador semântico garantir isso. Uma maneira de lidar com isso é construir o grafo de tipos e verificar se há ciclos.

Além disso, Cool tem algumas restrições com relação a herdar de classes básicas (vide manual). Também é um erro se uma classe herdar de uma classe que não está definida.

O esqueleto do projeto inclui definições apropriadas para todas as classes básicas. Você irá precisar incorporar essas classes na hierarquia de herança.

Nós sugerimos que você divida a sua análise semântica em dois componentes. Primeiro, verifique se o grafo de dependências está bem definido, isto é, todas as restrições de herança estão satisfeitas. Se o grafo de herança não está bem definido, é aceitável abortar a compilação (após exibir uma mensagem de erro apropriada, claro!). Por último, verifique todas as outras condições semânticas. É muito mais fácil implementar esse segundo componente tendo um grafo de herança válido.

5 Nomes e Escopo

Uma grande parte de qualquer analisador semântico é lidar com nomes e escopo. O problema em específico é determinar qual declaração está em efeito para cada uso de um identificador, especialmente quando nomes podem ser reutilizados. Por exemplo, se **i** é declarado em duas expressões let, uma aninhada dentro da outra, então quando **i** é referenciado, a semântica da linguagem define qual declaração está em efeito. É trabalho do analisador semântico manter o registro de qual declaração um nome se refere.

Uma *tabela de símbolos* é uma estrutura de dado conveniente para administrar nomes e escopos. Você pode utilizar a nossa implementação de uma tabela de símbolos no seu projeto. Ela oferece métodos para entrar, sair e aumentar escopos conforme necessário. Você também é livre para implementar a sua própria tabela de símbolos, se julgar mais conveniente.

Além do identificador **self**, que é implicitamente bound em toda classe, existem quatro maneiras em que nomes podem ser bound a objetos em Cool:

- Definição de atributos;
- Parâmetros formais de um método;
- Expressões let;

- Branches de cases.

Em adição a nome de objetos, também existem nomes de classes e métodos. É um erro usar qualquer nome que não tem uma declaração correspondente. Neste caso, entretanto, o analisador semântico *não* deve abortar a compilação após identificar este erro. Lembre-se que classes, métodos e atributos não precisam ser declarados antes do uso. Pense como isso afeta sua análise.

6 Verificação de Tipos

Verificação de tipos é uma outra função importantíssima de um analisador semântico. O analisador semântico deve verificar se tipos válidos são declarados onde são necessários. Por exemplo, o tipo de retorno de métodos precisa ser declarado. Usando essa informação, o analisador semântico precisa também verificar que toda expressão tem um tipo válido de acordo com as regras de tipagem da linguagem. As regras de tipagem de Cool são discutidas em detalhe no manual.

Uma tarefa difícil é decidir o que fazer quando uma expressão não tem um tipo válido de acordo com as regras de tipagem. Primeiro, uma mensagem de erro deve ser exibida com o número da linha e uma descrição do que deu errado. É relativamente fácil gerar mensages de erro informativas na análise semântica , pois geralmente o erro é óbvio. Nós esperamos que você gere mensagens de erro informativas.

Espera-se que o analisador semântico tente se recuperar quando identificar erros, mas não esperamos que ele consiga se recuperar em casos de erros em cascata. Um mecanismo simples de recuperação é atribuir o tipo Object a expressões que, de outra forma qualquer, não conseguiriam obter um tipo. Esse método foi utilizado no coolc.

7 Interface da geração de código

Para o analisador semântico funcionar corretamente com o resto do compilador coolc, alguns cuidados devem ser tomados para se adequar a interface com o gerador de código. Para cada nó de expressão, seu campo type deve ser atribuído ao Symbol nomeando o tipo inferido pelo seu verificador de tipos. Esse Symbol deve ser o resultado do método add_string de idtable. A expressão especial no_expr deve ser associada ao tipo No_type, que é um símbolo predefinido no esqueleto do projeto.

8 Saída Esperada

Para programas incorretos, a saída do seu analisador semântico são mensagens de erros. É esperado que se recupere de todos os errados exceto por hierarquias de classes mal formadas. Assumindo que a herança de classes esteja bem formada, o verificador semântico deve capturar e reportar todos os erros semânticos do programa. Suas mensagens de erro não precisam ser idênticas as do **coolc**.

Nós oferecemos para vocês o seguinte método pra reportar erros: **ostream& ClassTable::semant_error(Class_)**. Essa rotina pega um nó Class_ e retorna uma stream de saída que você pode usar para escrever mensagens de erro. Como o parser garante que nós Class_/class_ guardam o arquivo em que a classe foi definida (lembre-se que definições de classes não podem ser divididas entre arquivos), o número da linha da mensagem de erro pode ser obtido através do nó da AST onde o erro foi detectado e o nome do arquivo da classe.

Para programas corretos, a saída é uma AST com anotação de tipos.

9 Testando o analisador semântico

Você irá executar seu analisador semântico usando mysemant, um shell script que "integra" o análisador semântico com o parser e scanner. Perceba que mysemant recebe uma flag -s para debugar o analisador; utilizar essa flag meramente seta a variável global `semant_debug`. Pode ser útil caso queira escrever rotinas para auxiliar no debugging.

De maneira similar a fase de parsing, vocês devem executar make e make semant, para compilar o analisador semântico, gerando o mysemant e o mycoolc.

Assim que estiver confiante que seu analisador semântico está funcionando, tente executar mycoolc para invocar seu analisador junto com as outras etapas do compilador. Você deve testar esse compilador em entradas boas e ruins para verificar se está tudo funcionando. Lembre-se, bugs na análise semântica podem manifestar-se na geração de código ou somente quando o programa é executado com o spim.

10 Submissão

Vocês devem enviar os arquivos cool-tree.h, semant.cc e semant.h em um zip.