Trabalho Prático IV
Segunda, Novembro 21, 2016

1 Introdução

Nesse trabalho, você irá implementar a semântica estática de Cool. Você irá usar a Árvore Sintática Abstrata (AST) contruída pelo parser para verificar se um programa está em conformidade com a especificação de Cool. Seu componente estático semântico deve rejeitar programas incorretos; para programas corretos o componente deve agregar informações que serão utilizadas pelo gerador de código. A saída do analisador semântico será uma AST anotada para ser usada na geração de código.

Esse trabalho prático tem muito mais espaço para decisões de implementação que os trabalhos anteriores. Seu programa está correto se ele valida programas Cool de acordo com a especificação. Não há um único jeito correto de fazer o trabalho mas certamente há jeitos incorretos. Há algumas boas práticas que podem tornar sua vida mais fácil e iremos tentar te convencer de utilizá-las. Entretanto, o modo como você irá fazer é uma decisão sua. Qualquer que seja a sua implementação, esteja preparado para justificar e explicar a sua solução.

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

Existe muita informação nessa especificação e você irá precisar entender tudo para escrever um analisador semântico que funcione. *Por favor, faça uma leitura completa da especificação.*

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

1. Checar todas as classes e construir um grafo de herança
2. Checar se o grafo está bem formado
3. Para cada classe:
   (a) Atravessar a AST, agregando todas as declarações visíveis em uma tabela de símbolo
   (b) Validar se os tipos das expressões estão corretos
   (c) Anotar a AST com os tipos

2 Arquivos e Diretórios

Para começar, crie um diretório onde você deseja fazer o trabalho e execute o comando abaixo *dentro do diretório*:

```
make -f /home/prof/renato/cool/student/assignments/PA4/Makefile
```

Como de costume, têm vários arquivos que são usados no trabalho que estão como um elo simbólico no seu diretório ou são incluídos via /home/prof/renato/cool/student/assignments/PA4. **Não** modifique esses arquivos. Quase todos os arquivos foram descritos nos trabalhos anteriores. Veja as instruções no arquivo README.

2.1 C++ Version

Lista de arquivos que você pode querer modificar.

- cool-tree.h
  Esse arquivo é onde se coloca as suas extensões feitas nos nós da AST. É provável que você queira inserir declarações adicionais mas **não** modifique as declarações existentes.

- 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 implementação da **ClassTable** para representar o grafo de herança. Você decide se vai usar ou ignorar.

  O analisador semântico é invocado pela chamada do método **semant()** da classe **program_class**. A declaração de classe para **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. Implemente declarações adicionais que você precisar aqui e não em cool-tree.h.

- good.cl and bad.cl
  Esses arquivos são para testar algumas características semânticas. Você deve adicionar testes para garantir que good.cl exercite o máximo de combinações semânticas possíveis e que bad.cl exercite o máximo de erros semânticos possíveis. Não é possível exercitas todas as combinações em um arquivo só; você só é responsável por exercitar uma cobertura razoável. Explique os seus testes nesses arquivos e coloque comentários gerais no README.

- README
  Nesse arquivo irá conter a parte escrita do seu trabalho. Para esse trabalho é crítico que você explique suas decisões de implementação, como seu código está estruturado e porque você acredita que a sua implementação está correta. É parte do trabalho explicar o que você fez assim como comentar seu código. O README é muito importante para esse trabalho porque o README é o guia para compreender o seu trabalho.

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 agregar informações dela. Esse estilo algorítmico—caminhamento recursivo de uma estrutura complexa de árvore— é muito importante, porque é um jeito bastante natural de definir computações em ASTs.

Sua tarefa de programação para esse trabalho é (1) atravessar a árvore, (2) avaliar vários pedaços de informação que você quiser adquirir da árvore 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 de classes de dependências. Um requisito típico da maioria das linguagens com herança é que o grafo de herança seja acíclico. É função do seu verificador semântico de verificar e exigir esse requerimento. Um jeito fácil de fazer isso é construir uma representação de um grafo de tipos e procurar por ciclos.

Ainda mais, Cool possui restrições em herdar as classes básicas (veja no manual). Também é um erro se uma classe A herda da classe B mas B não está definida.

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

Nós sugerimos que você divida a sua análise semântica em dois componentes menores. 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!).

Segundo, verifique todas as outras condições semânticas. É muito mais fácil implementar esse segundo componente sabendo que o grafo de herança é válido.

5 Nomes e Escopo

Uma grande parte de qualquer analisador semântico é administrar os nomes. 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 verificador semântico manter o registro de qual declaração um nome se refere.

Como discutido em classe, 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. Nossa implementação oferece métodos para entrar, sair e aumentar escopos como necessário. Você também é livre para implementar a sua própria tabela de símbolos, claro.

Além do identificador **self**, que é implicitamente vinculado com toda classe, existem quatro maneiras de inserir o nome de um objeto em Cool:

- Definições de atribuição;
- Parâmetros formais de um método;
- Expressões let;
- Branches de case.

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. Nesse caso, entretanto, o analisador semântico *não* deve abortar a compilação após identificar esse 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 outra função magna de um analisador semântico. O analisador semântico deve verificar se tipos válidos são declarados quando necessário. 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 em acordo com as regras de tipagem. 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. 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 informatias na análise semântica porque geralmente o erro é óbvio. Nós esperamos que você gere mensagens de erro informativas. Por último, o analisador semântico deve ser capaz de se recuperar e continuar.

Nós esperamos que seu analisador semântico consiga se recuperar mas nós não esperamos que ele consiga evitar erros em cascata. Um mecanismo simples de recuperação é designar o tipo Object para qualquer expressão que não for possível designar 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 na interface com o gerador de código. Nós adotamos uma interface bastante simples para evitar limitar sua criatividade na análise semântica. Entretanto, há uma única coisa a se fazer. Para cada nó de expressão, seu campo type deve ser atribuído ao Symbol nomeando o tipo inferido pelo seu verificador de tipo. 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 erro. É esperado que se recupere de todos os errados exceto por hierarquias de classes mal formadas. É esperado também produzir erros completos e informativos. Assumindo que a herança de classes está bem formada, o verificador semântico deve capturar e reportar todos os erros semânticos do programa.

Suas mensagens de erro não precisam ser idênticas aquelas de **coolc**.

Nós oferecemos pra você alguns métodos de reportar erros **ostream& ClassTable::semant_error(Class_)**.

Essa rotina pega um nó Class_ e retorna um 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 dividas 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 envelope.

Para programas corretos, a saída é uma AST com anotação de tipos. A nota será definida se o seu analisador semântico anota as ASTs com tipos e se funciona integrado com o gerador de código de **coolc**.

9 Testando o analisador semântico

Você precisará de um scanner e um parser funcionando para testar seu analisador semântico. Você pode usar os seus ou os de coolc. Por padrão, as fases de coolc são utilizadas; para modificar isso, substitua os executáveis de lexer e parser com os seus executáveis. Recomendamos que utilize o seu analisador semântico com o scanner e o parser de coolc porque é o que iremos utilizar para definir a sua nota.

Você irá executar seu analisador semântico usando mysemant, um scrip shell que "cola" a análise com o parser e o scanner. Perceba que mysemant leva uma flag -s para debugar o analisador; utilizar essa flag meramente causa `semant_debug` (variável global para exibir debug) seja setado. Adicionar código extra pra ajudar no debug é por sua parte. Dê uma olhada no README para mais detalhes.

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 Considerações Importantes

A análise semântica é o maior componente do compilador de longe. A nossa solução tem aproximadamente 1300 linhas de código C++ bem documentado. Você achará essa tarefa mais fácil se desenhar a sua implementação antes de codificar. Pergunte a si mesmo:

- Quais requisitos eu preciso verificar?
- Quando eu preciso verificar um requisito?
- Quando é necessário alguma informação dos meus requisitos?
- Qual é a informação necessária para os meus requisitos?

Se você conseguir responder essas questões para cada requisito de Cool, implementar a solução vai ser como andar numa reta.

11 Submissão Final

Tenha certeza de completar os itens abaixo antes de submeter para evitar penalidades:

- ☐ Inclua sua documentação no README.
- ☐ Inclua seus casos de teste que devem passar na análise semântica dentro de good.cl e os testes que não passam em bad.cl
- ☐ Tenha certeza que todo seu código para a análise semântica está em
  - cool-tree.h, semant.h, and semant.cc