Trabalho Prático N.04 (Valor: 10 pontos)
Entrega: Conforme Tarefa no Canvas

1 Introdução

Neste trabalho, você implementará a análise semântica (estática) da linguagem *Cool*. Você deverá usar as árvores de sintaxe abstrata (AST) construídas pelo *parser* para verificar se um programa está em conformidade com a especificação da linguagem *Cool*. Seu analisador semântico deve rejeitar programas errados; enquanto que, para programas corretos, ele deve reunir/produzir certas informações para serem usadas pelo gerador de código. A saída do analisador semântico será uma AST anotada para uso pelo gerador de código.

Nesta tarefa, você terá muito mais espaço para tomar decisões de design do que teve nas anteriores. Sua implementação estará correta desde que seja capaz verificar se programas escritos na linguagem *Cool* se encontram em conformidade com a especificação formal da mesma. Não existe uma maneira "certa" e única de se realizar essa tarefa, mas certamente existem maneiras erradas! Acredita-se que existem várias práticas padronizadas que podem facilitar a obtenção de uma implementação correta. Nas aulas teóricas, o objetivo principal foi transmitir essas informações a você. No entanto, o resultado final vai depender muito de você. Portanto, para tudo o que você decidir fazer, esteja preparado para se justificar e explicar a solução adotada.

Você precisará consultar as regras de tipagem, regras de escopo de identificador e outras restrições da linguagem *Cool*, que se encontram definidas no *Manual de Referência de Cool*. Você também precisará adicionar métodos e membros de dados às definições de classe AST para esta fase. As funções que o pacote em árvore fornece estão documentadas junto ao material de *Suporte para Implementação da Linguagem Cool* (*Tour of Cool Support Code*).

Há muitas informações nesta especificação de trabalho e você precisa da maioria delas para escrever um analisador semântico que funcione corretamente. *Leia essa especificação detalhadamente!* Em uma descrição de alto nível, seu verificador semântico precisará executar as seguintes tarefas principais:

1. Considerar cada uma das classes contidas no programa e construir um grafo de herança;
2. Verificar se o grafo é bem formado;
3. Para cada classe:
   (a) Percorrer a AST, reunindo todas as declarações visíveis em uma tabela de símbolos;
   (b) Verificar cada expressão para garantir a correção dos tipos envolvidos; e
   (c) Anotar a AST com informações sobre tipos.

Esta lista de tarefas não é exaustiva; cabe a você implementar fielmente a especificação detalha no manual.

Você pode trabalhar em grupo de até 05 pessoas para esta tarefa (grupos maiores não serão aceitos em nenhuma hipótese).

2 Arquivos e Diretórios

Para começar, crie um diretório onde você deseja fazer o desenvolvimento de seu trabalho (por exemplo, `PA4`) e execute um dos seguintes comandos *nesse diretório*. Para a versão em C++ do trabalho, você deve digitar:

```
cd PA4
make -f /var/tmp/cool/assignments/PA4/Makefile
```

Enquanto que para a versão em Java, digite:

```
cd PA4
make -f /var/tmp/cool/assignments/PA4J/Makefile
```

(observe o **"J"** no nome do caminho).

Este comando irá copiar alguns arquivos para o seu diretório. Alguns dos arquivos serão copiados como "somente leitura" (usando links simbólicos). Você não deve alterar esses arquivos. Na realidade, se você criar e modificar cópias particulares desses arquivos, poderá terminar por achar impossível concluir o trabalho. Veja instruções mais detalhadas no arquivo README.

A seguir, descreve-se os arquivos mais importantes para cada versão do projeto.

2.1 Versão em C++

Esta é uma lista dos arquivos que você pode querer modificar.

- cool-tree.h
  Esse arquivo é onde as extensões definidas por você para os nós da árvore de sintaxe abstrata (AST) são colocadas. Você provavelmente precisará adicionar declarações adicionais, mas **não** modifique as declarações já existentes.

- semant.cc
  Este é o arquivo principal para sua implementação da fase de análise semântica. Ele contém alguns símbolos predefinidos para sua conveniência e o início de uma implementação de **ClassTable** para representar o grafo de herança. Você pode optar por usá-los ou ignorá-los.

  O analisador semântico é invocado por meio da chamada ao método **semant()** da classe **program_class**. A declaração de classe para **program_class** está em cool-tree.h. Quaisquer declarações de método adicionadas a cool-tree.h devem ser implementadas neste arquivo.

- semant.h
  Este arquivo é o arquivo de cabeçalho para semant.cc. Você deve adicionar neste arquivo quaisquer declarações adicionais necessárias (que não em estejam em cool-tree.h).

- good.cl e bad.cl
  Esses arquivos servem testar alguns recursos semânticos. Você deve adicionar testes para garantir que good.cl realize o maior número possível de combinações semânticas legais e que bad.cl cometa o maior número possível de erros semânticos. Não é possível exercitar todas as combinações possíveis em um arquivo; você é responsável apenas por obter uma cobertura razoável. Explique seus testes nesses arquivos e coloque quaisquer comentários gerais no arquivo README.

- README
  . Este arquivo conterá anotações sobre a realização de sua tarefa. Para essa atribuição, é fundamental que você explique as decisões de design, como seu código está estruturado e os motivos que levam você a acreditar que o design é bom (ou seja, razões para sua implementação estar correta e robusta). Faz parte da tarefa explicar tais decisões por meio de texto e também comentar seu código. Arquivos README inadequados serão penalizados mais fortemente nesta atribuição, pois o README é a principal fonte de informações que se tem para entender seu código final.

2.2 Versão em Java

Esta é uma lista dos arquivos que você pode querer modificar.

- cool-tree.java
  Este arquivo contém as definições para os nós AST e é o arquivo principal para sua implementação da fase de análise semântica. Você precisará adicionar o código para sua fase de análise semântica neste arquivo. O analisador semântico é invocado por meio da chamada ao método **semant()** da classe **programa**. **Não** modifique as declarações já existentes.

- ClassTable.java
  Esta classe é um espaço reservado para alguns métodos úteis (incluindo relatórios de erros e inicialização de classes básicas). Você pode aprimorá-lo para uso em seu analisador.

- TreeConstants.java
  Este arquivo define algumas constantes simbólicas que são muito úteis.

- good.cl e bad.cl
  Esses arquivos servem para testar alguns recursos semânticos. Você deve adicionar testes para garantir que good.cl realize o maior número possível de combinações semânticas legais e que bad.cl cometa o maior número possível de erros semânticos. Não é possível exercitar todas as combinações possíveis em um arquivo; você é responsável apenas por obter uma cobertura razoável. Explique seus testes nesses arquivos e coloque quaisquer comentários gerais no arquivo README.

- README
  Este arquivo conterá anotações sobre a realização de sua tarefa. Para essa atribuição, é fundamental que você explique as decisões de design, como seu código está estruturado e os motivos que levam você a acreditar que o design é bom (ou seja, razões para sua implementação estar correta e robusta). Faz parte da tarefa explicar tais decisões por meio de texto e também comentar seu código. Arquivos README inadequados serão penalizados mais fortemente nesta atribuição, pois o README é a principal fonte de informações que se tem para entender seu código final.

3 Caminhamento na Árvore de Sintaxe Abstrata

Como resultado do trabalho prático anterior, seu *parser* cria árvores de sintaxe abstratas (ASTs). O método **dump_with_types**, definido na maioria dos nós da AST, ilustra como percorrer a AST e coletar informações dela. Esse estilo solução algorítmica – envolvendo um caminhamento recursivo de uma estrutura de árvore complexa – é muito importante, porque representa é uma maneira muito natural de se organizar tarefas e processamentos em ASTs.

Sua implementação para este trabalho deve: (1) percorrer a árvore; (2) gerenciar várias informações que você recolher da árvore; e (3) usar essas informações para validar a semântica da linguagem *Cool*. Um caminhamento completo da AST é chamado de "passo". Você provavelmente precisará fazer pelo menos dois passos pela AST para verificar tudo.

Você provavelmente precisará anexar informações personalizadas aos nós da AST. Para fazer isso, você deve modificar cool-tree.h (na versão em C++) ou cool-tree.java (na versão em Java) diretamente. Na versão C++, as implementações de métodos que você deseja adicionar devem SER codificadas em semant.cc.

4 Herança

Os relacionamentos de herança especificam um grafo direcionado de dependências entre classe. Um requisito típico da maioria das linguagens com herança é que o grafo de herança seja acíclico. Cabe ao

seu verificador semântico impor esse requisito. Uma maneira bastante fácil de fazer isso é construir uma representação do grafo de tipos e verificar a existência de ciclos no mesmo.

Além disso, a linguagem *Cool* possui restrições de herança das classes básicas (consulte o manual). Também é um erro se a classe A herda da classe B, mas a classe B não está definida.

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

Sugerimos que você divida sua fase de análise semântica em dois componentes menores. Primeiro, verifique se o grafo de herança está bem definido, o que significa que todas as restrições relativas à herança são atendidas. Se o grafo de herança não estiver bem definido, é aceitável interromper a compilação (após a impressão de mensagens de erro apropriadas, é claro!). Segundo, verifique todas as outras condições semânticas. É muito mais fácil implementar esse segundo componente se já se conhece o grafo de herança e se o mesmo é legal (isto é, bem formado).

5 Nomes e Escopo

Uma parte importante de qualquer verificador semântico é o gerenciamento de nomes. O problema específico é determinar qual declaração está em vigor para cada uso de um identificador, especialmente quando os nomes podem ser reutilizados. Por exemplo, se **i** for declarado em duas expressões let, uma aninhada na outra, em qualquer lugar em que **i** for referenciado, a semântica da linguagem especificará qual declaração está em vigor. É tarefa do verificador semântico acompanhar a qual declaração um determinado nome se refere.

Conforme discutido na aula, uma *tabela de símbolos* é uma estrutura de dados conveniente para gerenciar nomes e escopo. Você pode usar implementação disponibilizada (no código de suporte) de tabelas de símbolos para seu projeto. Essa implementação fornece métodos para inserir, sair e aumentar escopos, conforme necessário. Você também pode implementar sua própria tabela de símbolos, é claro.

Além do identificador **self**, que está implicitamente vinculado em todas as classes, há quatro maneiras pelas quais um nome de objeto pode ser introduzido na linguagem *Cool*:

- definições de atributos;
- parâmetros formais de métodos;
- expressões let;
- ramos (*branches*) de instruções case.

Além dos nomes dos objetos, também existem nomes de métodos e nomes de classes. É um erro usar qualquer nome que não tenha uma declaração correspondente. Nesse caso, no entanto, o analisador semântico não deve abortar a compilação depois de descobrir esse erro. Lembre-se de que nem classes, nem métodos e nem atributos precisam ser declarados antes do uso. Pense em como isso afetará sua análise!

6 Verificação de Tipos

A verificação de tipo é outra função importante do analisador semântico. O analisador semântico deve verificar se tipos válidos são declarados onde necessário. Por exemplo, os tipos de retorno de métodos devem ser declarados. Usando essas informações, o analisador semântico também deve verificar se toda expressão tem um tipo válido de acordo com as regras de tipo. As regras de tipo são discutidas em detalhes no *Manual de Referência de Cool* e nos *slides* usados nas aulas teóricas do curso.

Uma questão difícil é determinar o que fazer se uma expressão não tiver um tipo válido de acordo com as regras. Primeiro, uma mensagem de erro deve ser impressa com o número da linha e uma descrição do que deu errado. É relativamente fácil enviar mensagens de erro informativas na fase de análise semântica,

porque geralmente é óbvio qual é o erro. Esperamos que você forneça mensagens de erro informativas. Segundo, o analisador semântico deve tentar recuperar e continuar.

Espera-se que seu analisador semântico se recupere, mas não se espera que ele evite erros em cascata. Um mecanismo de recuperação simples é atribuir o tipo Object a qualquer expressão que não possa receber um tipo (este método foi usado na implementação padrão do compilador coolc).

7 Interface com o Gerador de Código

Para que o analisador semântico funcione corretamente com o restante do compilador coolc, deve-se tomar cuidado para se utilizar adequadamente a interface com o **gerador de código**. Adotou-se deliberadamente uma interface muito simples e ingênua para evitar restringir quaisquer impulsos criativos que você venha a ter na análise semântica. No entanto, há uma coisa que você necessariamente deve fazer. Para cada nó de expressão, seu campo type deve ser "*setado*" como o Symbol nomeando o tipo inferido pelo seu verificador de tipos. Este Symbol deve ser o resultado da chamada do método add_string (na versão em C++) ou addString (na versão em Java) pertencente a idtable. A expressão especial no_expr deve receber o tipo No_type, que é um símbolo predefinido no esqueleto do projeto.

8 Saída Esperada

Para programas incorretos, a saída da análise semântica são mensagens de erro. Você deve se recuperar de todos os erros, exceto para hierarquias de classe mal formadas. Também é esperado que você produza erros completos e informativos. Supondo que a hierarquia de herança seja bem formada, o verificador semântico deve capturar e relatar todos os erros semânticos no programa.

Suas mensagens de erro não precisam ser idênticas às do compilador padrão **coolc**. Fornecemos a você métodos simples de relatório de erros **ostream& ClassTable::semant_error(Class_)** (na versão em C++) e **PrintStream ClassTable.semantError(class_)** (na versão em Java).

Essa rotina utiliza um nó Class_ (na versão em C++) ou class_ (na versão em Java) e retorna um fluxo de saída que você pode usar para gravar mensagens de erro. Como o *parser* garante que os nós Class_ / class_ armazenem o arquivo no qual a classe foi definida (lembre-se de que as definições de classe não podem ser divididas entre arquivos), o número da linha da mensagem de erro pode ser obtido a partir do nó da AST em que o erro foi detectado e o nome do arquivo da classe envolvente.

Para programas corretos, a saída é uma árvore de sintaxe abstrata com anotação de tipo. Você será avaliado se sua fase semântica anota corretamente ASTs com tipos e se sua fase semântica funciona corretamente com o gerador de código padrão do compilador **coolc**.

9 Testando o Analisador Semântico

Você precisará de um *scanner* e um *parser* funcionais para testar seu analisador semântico. Você pode usar seus próprios *scanner*/*parser* (desenvolvidos nos trabalhos anteriores) ou os scanner/parser do compilador padrão coolc. Por padrão, as fases coolc já são usadas; para mudar isso, substitua o executável lexer e/ou parser (que são links simbólicos no diretório do projeto) pelo sua própria implementação de *scanner* e/ou *parser*. Mesmo se você usar seu próprio *scanner* e/ou *parser*, é aconselhável testar seu analisador semântico com o *scanner* e o *parser* do compilador padrão de coolc pelo menos uma vez, porque seu analisador semântico será avaliado usando essas implementações.

Você executará seu analisador semântico usando mysemant, um *shell script* que "cola" o seu analisador semântico com o *scanner* e o *parser*. Observe que mysemant usa um *flag* -s para depurar o analisador; o uso desse sinalizador apenas faz com que `semant_debug` (uma variável global na versão em C++ e um campo estático da classe `Flags` na versão em Java) seja definida(o). Adicionar o código real para produzir

informações úteis de depuração é parte de sua tarefa. Veja mais informações no arquivo README para detalhes sobre essa parte do projeto.

Quando tiver certeza de que seu analisador semântico está funcionando, tente executar mycoolc para chamar seu analisador junto com outras fases do compilador. Você deve testar esse compilador em entradas boas e ruins para ver se tudo está funcionando. Lembre-se de que erros no analisador semântico podem se manifestar no código gerado ou somente quando o programa compilado é executado no simulador spim.

10 Observações

A fase de análise semântica é de longe o maior componente do compilador até agora. A solução da implementação padrão possui aproximadamente 1300 linhas de código em C++ (bem documentadas). Você encontrará a tarefa mais fácil se demorar algum tempo para projetar o verificador semântico antes da codificação. Pergunte a si mesmo:

- Quais requisitos eu preciso verificar?
- Quando preciso verificar um requisito?
- Quando as informações necessárias para verificar um requisito são geradas?
- Onde estão as informações necessárias para verificar um requisito?

Se você puder responder a essas perguntas para cada aspecto da linguagem *Cool*, a implementação de uma solução deve ser bem simples e direta.

11 O que entregar

Certifique-se de concluir os seguintes itens antes de enviar para evitar penalidades na avaliação de seu trabalho.

- Modificar o final do arquivo `README`. Parte dessa tarefa é preencher o `README` com anotações sobre o seu projeto.
- Inclua os casos de teste que devem passar (positivos) no analisador semântico em good.cl e os casos de teste (negativos) que devem fazer com que o analisador semântico emita um erro em bad.cl.
- Assegure-se de que todo o seu código para o analisador semântico está em nos seguintes arquivos:
  - cool-tree.h, semant.h e semant.cc para a versão em C++ version; ou
  - cool-tree.java, ClassTable.java e TreeConstants.java para a versão em Java.
- Compactar e codificar o subdiretório `PA4`, por exemplo da seguinte forma:

```
tar cvzf PA4.tar.gz PA4
uuencode PA4.tar.gz PA4.tar.gz > PA4.u
rm PA4.tar.gz
```

- Entregar o arquivo `PA4.u` via **Canvas**.