Trabalho Prático N.03 (Valor: 10 pontos)
Entrega: Conforme Tarefa no Canvas

1 Introdução

Neste trabalho, você irá implementar um *parser* para a linguagem *Cool*. Esta tarefa faz uso de duas ferramentas: um gerador de *parser* (a ferramenta para C++ se chama Bison; enquanto a ferramenta em Java é chamada CUP) e um pacote para manipulação de árvores.

A saída de seu parser deve ser uma árvore de sintaxe abstrata (*Abstract Syntax Tree* – AST) e você deverá construir esta AST por meio da especificação de ações semânticas junto ao gerador de *parser*.

Você certamente irá precisar fazer referência e utilizar a estrutura sintática da linguagem *Cool*, que pode ser encontrada na Figura 1 do *Manual de Referência de Cool*. A documentação de bison and CUP pode ser encontrada *online*. A versão em C++ do pacote para manipulação de árvores se encontra descrita junto ao material de *Suporte para Implementação da Linguagem Cool* (*Tour of Cool Support Code*). Já a documentação para versão em Java do pacote para manipulação de árvores se encontra disponível em `https://web.stanford.edu/class/archive/cs/cs143/cs143.1112/javadoc/cool_ast/index.html?package-tree.html`. Você ira precisar de consultar informações sobre o pacote de manipulação de árvores na realização deste trabalho e dos próximos.

Há muita informação nesta especificação e você precisa saber a maior parte para conseguir escrever um *parser* funcional. **Por favor, leia atentamente toda essa especificação**.

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, `PA3`) e execute um dos seguintes comandos *nesse diretório*. Para a versão em C++ do trabalho, você deve digitar:

```
cd PA3
make -f /var/tmp/cool/assignments/PA3/Makefile
```

Enquanto que para a versão em Java, digite:

```
cd PA2
make -f /var/tmp/cool/assignments/PA3J/Makefile
```

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

Este comando irá copiar alguns arquivos para o seu diretório. Alguns dos arquivos serão copiados no modo "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.

Os arquivos que você precisará modificar são:

- cool.y (na versão em C++) / cool.cup (na versão em Java)
  Este arquivo contém um esqueleto de uma descrição do *parser* para a linguagem *Cool*. A seção de declaração está quase completa, mas você precisará adicionar declarações de tipo adicionais para novos não-terminais que você introduzir. Já lhe serão fornecidas declarações de nomes e de tipos para os terminais. Você também pode precisar adicionar declarações de precedência. A seção de regras, no

entanto, se encontra bastante incompleta. Já lhe foram fornecidas algumas partes de algumas regras. Então, você não precisa modificar esse código para obter uma solução funcional (*parser*), mas será se desejar, você é livre :) . No entanto, você não deve assumir que qualquer regra específica esteja completa.

- good.cl and bad.cl
  Esses arquivos permitem testar alguns recursos da gramática da linguagem *Cool*. Você deve adicionar testes para garantir que good.cl realize testes de toda e qualquer "construção legal" da gramática e que bad.cl realize testes do maior número possível de erros de análise em um único arquivo. Você deve explicar tanto seus testes nesses arquivos e como colocar comentários gerais no arquivo README.

- README
  Como de costume, esse arquivo conterá as anotações sobre a realização de sua tarefa. Explique suas decisões de design, seus casos de teste e por que você acredita que seu programa é correto e robusto. Faz parte da tarefa explicar as coisas nesse texto, bem como comentar o seu código.

3 Testando o *Parser*

Você irá precisar de um *scanner* (analisador léxico) funcionando corretamente para ser capaz de testar seu *parser*. Você pode usar tanto sua própria implementação de *scanner* (feito no trabalho anterior) como a versão oficial fornecida junto com o compilador coolc. Como padrão, o *scanner* da versão oficial do compilador coolc é utilizado; para modificar esse comportamento basta substituir o executável lexer (presente no diretório do projeto por meio de uma *ligação simbólica* ou atalho) pelo seu próprio *scanner* (desenvolvido anteriormente). Não assuma automaticamente que o *scanner* utilizado está livre de erros – erros ocultos (não identificados) no *scanner* podem causar problemas misteriosos no comportamento do seu *parser*.

Você deverá executar/testar seu *parser* por meio do "*shell script*" myparser que realizar uma chamada para o *scanner* seguida de uma chamada para o *parser*. Observe que myparser usa um "*flag*" -p para depurar o parser; esse "*flag*" faz com que muitas informações sobre o que o *parser* está fazendo sejam impressas na saída padrão. Além disso, tanto Bison quanto CUP produzem um listagem legível das tabelas sintáticas LALR(1) no arquivo cool.output. Examinar esse arquivo pode também ser útil para depurar a definição do seu *parser*.

Você deve testar seu compilador em entradas "*boas*" e "*ruins*" para ver se tudo está funcionando corretamente. Lembre-se, os erros na definição de seu parser podem se manifestar em qualquer lugar.

Seu *parser* será avaliado usando a versão oficial do analisador léxico (*scanner*) do compilador coolc. Assim, mesmo que você faça a maior parte do trabalho usando seu próprio *scanner*, você deve testar seu *parser* com a versão oficial do analisador léxico (*scanner*) do compilador coolc antes de entregar a tarefa.

4 Resultados do *Parser*

Você deve especificar um conjunto de ações semânticas que sejam capazes de construir uma Árvore Sintática Abstrata, ou *Abstrat Syntax Tree (AST)*. A raiz (e somente a raiz) da AST deve ser do tipo `program`. Para programas que forem analisados com êxito, a saída do `parser` é uma listagem da AST.

Para programas que possuem erros, a saída consiste das mensagens de erro do parser. Você já receberá uma rotina de informe de erros que imprime/exibe mensagens de erro em um formato padrão; por favor, não a modifique. Você não deve invocar esta rotina diretamente em suas ações semânticas; Bison/CUP realiza a invocação dela automaticamente quando um problema é detectado.

Seu parser só precisa trabalhar para programas contidos em um único arquivo – não se preocupe em compilar múltiplos arquivos.

5 Tratamento de Erros

Você deve usar o (pseudo) não-terminal `error` para adicionar capacidades de manipulação de erros ao seu *parser*. O objetivo de `error` é permitir que o *parser* continue operando mesmo após algum erro (na medida em que se possa antecipar sua ocorrência). Porém, saiba que essa abordagem não representa uma "solução universal" e seu *parser* pode ficar completamente confuso.

Veja a documentação do Bison/CUP sobre as formas de como melhor usar `error`. Em seu arquivo README, você deve descrever quais erros você tentou capturar. Além disso, seu arquivo de teste bad.cl deve ter algumas instâncias que ilustrem tais erros a partir dos quais seu *parser* pode se recuperar. Para receber uma avaliação total, seu parser deve se recuperar pelo menos das seguintes situações:

- Se houver um erro em uma definição de classe, mas a classe for finalizada corretamente e a próxima classe estiver sintaticamente correta, o seu *parser* deverá ser capaz de reiniciar nessa próxima definição de classe.
- Da mesma forma, seu *parser* deve se recuperar de erros na definição de:
  1. *features* (atributo estático ou dinâmico) de uma classe, passando para a próxima *feature*);
  2. uma "ligação" de uma variável em um comando `let`, passando para a próxima variável; e
  3. uma expressão dentro de um bloco `{...}`, continuando depois do `}`.

Não se preocupe excessivamente com os números de linha que aparecem nas mensagens de erro geradas pelo *parser*. Se o *parser* estiver funcionando corretamente, o número da linha geralmente será a linha em que ocorreu o erro. Para construções erradas quebradas em várias linhas, o número da linha provavelmente será a última linha da construção.

6 Pacote de Manipulação de Árvore

Há uma documentação extensa sobre a versão C++ do pacote para manipulação árvores a ser utilizado na implementação de AST para Cool junto ao material de *Suporte para Implementação da Linguagem Cool* (*Tour of Cool Support Code*). A documentação para a versão Java está disponível na página em `https://web.stanford.edu/class/archive/cs/cs143/cs143.1112/javadoc/cool_ast/index.html?package-tree.html`. Você certamente precisará da maioria dessas informações contidas nessa documentação para implementar um *parser* funcional.

7 Observações

Você pode usar declarações de precedência, mas apenas para expressões. Não deve usar declarações de precedência cegamente (ou seja, não trate um conflito reduzir/empilhar em sua gramática, adicionando regras de precedência até que ele desapareça).

A construção `let` na linguagem *Cool* introduz uma ambiguidade na linguagem (tente construir um exemplo se você não estiver convencido). O manual resolve a ambiguidade dizendo que uma expressão `let` se estende o mais à direita possível. A ambiguidade aparecerá em seu *parser* como um conflito reduzir/empilhar envolvendo as produções de `let`.

Esse problema tem uma solução simples, mas um pouco obscura. Não lhe será dito exatamente como resolvê-lo, mas você receberá uma boa dica.

No compilador oficial `coolc`, implementou-se a resolução do conflito reduzir/empilhar da construção `let` usando um recurso Bison/CUP que permite que precedência seja associada a quais quer produções (não apenas operadores). Consulte a documentação de Bison/CUP para obter maiores informações sobre como usar este recurso.

Como o compilador mycoolc usa "*pipes*" (redirecionamentos) para estabelecer a comunicação de um estágio com o próximo, qualquer caractere estranho produzido pelo seu *parser* pode causar erros; em particular, o analisador semântico pode não ser capaz de processar a AST que seu parser produzir.

8 Notas adicionais para a versão em C++

Se você estiver trabalhando na versão em Java, pule para a seção seguinte.

- Executar Bison no arquivo inicial do esqueleto produzirá alguns avisos sobre "*não terminais inúteis*" e "*regras inúteis*". Isso ocorre porque alguns dos não terminais e regras nunca serão usados, mas esses avisos *devem* desaparecer quando o seu *parser* estiver completo.

- Você deve declarar "tipos" em Bison para seus não terminais e terminais que possuírem atributos. Por exemplo, no esqueleto cool.y, você encontra a declaração:

  ```
  %type <program> program
  ```

  Esta declaração diz que o não terminal `program` tem tipo `<program>`.

  O uso da palavra "**tipo**" aqui é enganoso; o que essa declaração realmente significa é que o atributo para o não terminal `program` é armazenado no membro `program` da declaração `union` em cool.y, que possui tipo `Program`.

  Por meio da especificação de "*tipo*"

  ```
  %type <member_name> X Y Z ...
  ```

  você está instruindo o **Bison** que que os atributos dos não terminais (ou terminais) `X`, `Y`, and `Z` possuem o tipo apropriado para o membro denominado `member_name` da *união*.

  Todos os membros da *união* e seus tipos têm nomes semelhantes por escolha de projeto. É apenas uma coincidência no exemplo acima que o não terminal `program` tenha o mesmo nome de um membro da *união*.

  É fundamental que você declare os tipos corretos para os atributos dos símbolos gramaticais; uma falha em fazê-lo praticamente garante que o *parser* não funcionará. Você não precisa declarar tipos para símbolos da sua gramática que não possuam atributos.

  O verificador de tipos de compilador `g++` reclama se você usar os construtores de árvore com os parâmetros de tipos incorretos. Se você ignorar os avisos, o programa poderá falhar quando o construtor perceber que está sendo usado incorretamente. Além disso, Bison pode reclamar se você cometer erros de tipagem. Preste atenção a qualquer aviso. Não se surpreenda se seu programa falhar quando Bison ou `g++` derem mensagens de aviso.

9 Notas adicionais para a versão em Java

Se você estiver trabalhando na versão em C++, pule esta seção.

- Você deve declarar "tipos" em `CUP` para os seus não terminais e terminais que possuírem atributos. Por exemplo, no esqueleto que está no arquivo cool.cup, você encontra a seguinte declaração:

  ```
  nonterminal program program;
  ```

Esta declaração diz que o não terminal `program` possui "tipo" `program`.

É fundamental que você declare os tipos corretos para os atributos dos símbolos gramaticais; uma falha em fazê-lo praticamente garante que o seu *parser* não funcionará. Você não precisa declarar tipos de símbolos da sua gramática que não possuam atributos.

O verificador de tipos do compilador javac reclama se você usar os construtores de árvore com os parâmetros de tipos incorretos. Se você corrigir os erros com conversões frívolas, seu programa poderá lançar uma exceção quando o construtor perceber que está sendo usado incorretamente. Além disso, CUP pode reclamar se você cometer erros de tipagem.

10 O Que Entregar

Quando você estiver pronto para entregar a tarefa, digite make submit-clean no diretório em que você preparou seu trabalho. Essa ação removerá todos os arquivos desnecessários, como arquivos de objeto, arquivos de classe, *core dumps*, etc. Certifique-se, também, de ter editado corretamente o arquivo README antes de enviar seu trabalho. O código, a saída e anotações, quando confusas e pouco claras, terão um efeito negativo na sua nota. Aproveite bem seu tempo para explicar de forma clara (e concisa!) seus resultados.

Depois de concluído o trabalho, você deverá realizar a entrega dessa tarefa. Para tanto, você deve fazer as seguintes atividades:

1. Assegurar-se que que seu código esteja no arquivo `cool.flex` para a implementação em C++ (ou `cool.lex` para a implementação em Java) e de de que ele compila e funciona corretamente :) .

2. Modificar o final do arquivo `README`. Parte dessa tarefa é preencher o `README` com anotações sobre o seu projeto. Você deve explicar as decisões de design, explicar por que seu código está correto e por que seus casos de teste são adequados. Faz parte da tarefa explicar de forma clara e concisa, bem como comentar o seu código. Apenas altere o arquivo fornecido.

3. Compactar e codificar o subdiretório `PA3`, por exemplo da seguinte forma:

```
tar cvzf PA3.tar.gz PA3
uuencode PA3.tar.gz PA3.tar.gz > PA3.u
rm PA3.tar.gz
```

4. Entregar o arquivo `PA3.u` via **Canvas**.