Tarefa de Programação III

1. Introdução

Nesta tarefa, você escreverá um analisador para Cool. A atribuição faz uso de duas ferramentas: o gerador de parser (a ferramenta C++ é chamada bison) e um pacote para manipulação de árvores. A saída do seu analisador será uma árvore de sintaxe abstrata (AST). Você construirá este AST usando ações semânticas do gerador do analisador.

Você certamente precisará consultar a estrutura sintática de Cool, encontrada na Figura 1 do The Cool Reference Manual (assim como em outras partes). A documentação do bison está disponível no moodle. A versão C++ do pacote tree é descrita no folheto Tour of Cool Support Code. Você precisará das informações do pacote de árvore para esta e futuras atribuições.

Há muitas informações neste folheto, e você precisa saber a maior parte delas para escrever um trabalho analisador. Por favor, leia atentamente o folheto.

1.1 Instruções Adicionais

1. Você deve usar as máquinas Linux no laboratório de pós-graduação do DCC (máquinas *.grad).

   (a) Alguns exemplos são congo.grad, xingu.grad, eufrates.grad, jordao.grad, vermelho.grad, claro.grad, mucuri.grad, piracicaba.grad, niger.grad, paracatu.grad e doce.grad . (b) As tarefas de programação serão avaliadas em uma dessas máquinas, portanto, certifique-se de que seu envio funcione corretamente neste ambiente, caso contrário, poderá afetar sua nota.

2. Não serão toleradas cópias de colegas, ex-alunos ou qualquer outra fonte.

2 Arquivos e Diretórios

Para começar, crie um diretório no qual você deseja fazer a atribuição e execute um dos seguintes comandos nesse diretório. Você deve digitar

```
make -f /home/prof/renato/cool/student/assignments/PA3/Makefile
```

Este comando copiará vários arquivos para seu diretório. Alguns dos arquivos serão copiados somente leitura (usando links simbólicos). Você não deve editar esses arquivos. Na verdade, se você fizer e modificar cópias particulares desses arquivos, poderá achar impossível concluir a tarefa. Consulte as instruções no arquivo README. Os arquivos que você precisará modificar são:

cool.y (na versão C++)

Este arquivo contém um começo para uma descrição do analisador para Cool. A seção de declaração está quase completa, mas você precisará adicionar declarações de tipo adicionais para novos não terminais que você introduzir.
Demos a você nomes e declarações de tipo para os terminais. Você também pode precisar adicionar declarações de precedência. A seção de regras, no entanto, é bastante incompleta. Fornecemos algumas partes de algumas regras. Você não deve precisar modificar este código para obter uma solução funcional, mas é bem-vindo se quiser. No entanto, não assuma que qualquer regra em particular está completa.

good.cl e bad.cl Esses
arquivos testam alguns recursos da gramática. Você deve adicionar testes para garantir que o good.cl exerça todas as construções legais da gramática e que o bad.cl exerça tantos tipos de erros de análise quanto possível em um único arquivo. Explique seus testes nesses arquivos e coloque quaisquer comentários gerais no arquivo README.

README
Como de costume, este arquivo conterá a redação do seu trabalho. Explique suas decisões de projeto, seus casos de teste e por que você acredita que seu programa está correto e robusto. Faz parte da tarefa explicar as coisas em texto, bem como comentar seu código.

3 Testando o analisador

Você precisará de um scanner funcional para testar o analisador. Você pode usar seu próprio scanner (basta colocar de lexer bin na pasta de trabalho). Não assuma automaticamente que o scanner (qualquer um que você use!) está livre de bugs - bugs latentes no scanner podem causar problemas misteriosos no analisador.

Você executará seu analisador usando myparser, um script de shell que “cola” o analisador com o scanner. Observe que myparser recebe um sinalizador -p para depurar o analisador; usar este sinalizador faz com que muitas informações sobre o que o analisador está fazendo sejam impressas no stdout. Bison produz um dump legível das tabelas de análise LALR(1) no arquivo cool.output. Examinar esse dump é frequentemente útil para depurar a definição do analisador.

Você deve testar este compilador em entradas boas e ruins para ver se tudo está funcionando. Lembre-se, bugs em seu analisador podem se manifestar em qualquer lugar.

Seu analisador será avaliado usando nosso analisador léxico. Assim, mesmo que você faça a maior parte do trabalho usando seu próprio scanner, você deve testar seu analisador com o scanner coolc antes de entregar a tarefa.

4 Saída do analisador

Suas ações semânticas devem construir um AST. A raiz (e somente a raiz) do AST deve ser do tipo programa. Para programas que analisam com sucesso, a saída do analisador é uma listagem do AST.

Para programas com erros, a saída são as mensagens de erro do analisador. Fornecemos a você uma rotina de relatório de erros que imprime mensagens de erro em um formato padrão; Por favor, não o modifique.
Você não deve invocar essa rotina diretamente nas ações semânticas; bison o invoca automaticamente quando um problema é detectado.

Seu analisador só precisa funcionar para programas contidos em um único arquivo - não se preocupe em compilar vários arquivos.

5 Tratamento de erros

Você deve usar o pseudo-não terminal de erro para adicionar recursos de tratamento de erros no analisador. O propósito do erro é permitir que o analisador continue após algum erro antecipado. Não é uma panacéia e o analisador pode ficar completamente confuso. Consulte a documentação do bison para saber a melhor forma de usar o erro.
Em seu README, descreva quais erros você tenta detectar. Seu arquivo de teste bad.cl deve ter algumas instâncias que ilustram os erros dos quais seu analisador pode se recuperar. Para receber o crédito total, seu analisador deve se recuperar pelo menos nas seguintes situações:

Se houver um erro em uma definição de classe, mas a classe for encerrada corretamente e a próxima classe estiver sintaticamente correta, o analisador deverá ser capaz de reiniciar na próxima definição de classe.

Da mesma forma, o analisador deve se recuperar de erros nos recursos (passando para o próximo recurso), uma ligação let (passando para a próxima variável) e uma expressão dentro de um bloco {...}.

Não se preocupe excessivamente com os números de linha que aparecem nas mensagens de erro que seu analisador gera. Se o seu analisador estiver funcionando corretamente, o número da linha geralmente será a linha onde ocorreu o erro. Para construções errôneas divididas em várias linhas, o número da linha provavelmente será a última linha da construção.

6 O Pacote da Árvore

Há uma extensa discussão sobre a versão C++ do pacote tree para árvores de sintaxe abstrata Cool na seção Tour da documentação Cool. Você vai precisar da maioria dessas informações para escrever um trabalho analisador.

7 Observações

Você pode usar declarações de precedência, mas apenas para expressões. Não use declarações de precedência cegamente (ou seja, não responda a um conflito shift-reduce em sua gramática adicionando regras de precedência até que ele desapareça).

A construção Cool let 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 analisador como um conflito shift-reduce envolvendo as produções para let.

Este problema tem uma solução simples, mas um pouco obscura. Não diremos exatamente como resolvê-lo, mas daremos uma dica forte. No coolc, implementamos a resolução do conflito let shift-reduce usando um recurso bison que permite que a precedência seja associada a produções (não apenas a operadores).
Consulte a documentação do bison para obter informações sobre como usar esse recurso.

Como o compilador mycoolc usa pipes para se comunicar de um estágio para o próximo, quaisquer caracteres estranhos produzidos pelo analisador podem causar erros; em particular, o analisador semântico pode não ser capaz de analisar o AST que seu analisador produz.

8 Notas para a versão C++ da atribuição

A execução do bison no arquivo esqueleto inicial produzirá alguns avisos sobre “noterminais inúteis” e “regras inúteis”. Isso ocorre porque alguns dos não-terminais e regras nunca serão usados, mas eles devem desaparecer quando seu analisador estiver concluído.

Você deve declarar “tipos” de bisonte para seus não-terminais e terminais que tenham atributos. Por exemplo, no esqueleto cool.y está a declaração:

```
%type <programa> programa
```

Esta declaração diz que o programa não terminal tem o tipo <program>. O uso da palavra “tipo” é enganoso aqui; o que realmente significa é que o atributo para o programa não terminal é armazenado no membro do programa da declaração da união em cool.y, que tem o tipo Program. Ao especificar o tipo

%type <nome_membro> XYZ ...

você instrui o bison que os atributos dos não terminais (ou terminais) X, Y e Z têm um tipo apropriado para o nome do membro membro da união.

Todos os membros do sindicato e seus tipos têm nomes semelhantes por design. É uma coincidência no exemplo acima que o programa não terminal tenha o mesmo nome de um sindicalizado.

É fundamental que você declare os tipos corretos para os atributos dos símbolos gramaticais; a falha em fazê-lo praticamente garante que seu analisador não funcionará. Você não precisa declarar tipos para símbolos de sua gramática que não possuem atributos.

O verificador de tipo do g++ reclama se você usar os construtores de árvore com os parâmetros de tipo errados. Se você ignorar os avisos, seu programa poderá travar quando o construtor perceber que está sendo usado incorretamente. Além disso, o bisão pode reclamar se você cometer erros de tipo. Preste atenção a quaisquer avisos. Não se surpreenda se o seu programa travar quando bison ou g++ derem mensagens de aviso.