Trabalho Prático 3

1 Introdução

Neste trabalho você deverá implementar um analisador sintático (*parser*) para a linguagem `Cool`. Este trabalho faz uso de duas ferramentas: `bison`, que é um gerador de analisadores sintáticos, e um pacote para manipulação de árvores. O resultado de seu analisador sintático será uma Árvore de Sintaxe Abstrata (AST). Você deverá seguir a estrutura sintática da linguagem `Cool` apresentada na Figura 1 do documento *The Cool Reference Manual*. O pacote de manipulação de árvores é descrito no documento *A Tour of Cool Support Code*. Ambos os arquivos podem ser encontrados na pasta "doc", presente no arquivo "tp3-4.tar.gz", assim como a documentação da ferramente `bison`. **O que deve ser entregue**: os arquivos modificados mais o arquivo README com a documentação do trabalho.

2 Arquivos e Diretórios

Para começar, extraia o arquivo "tp3-4.tar.gz" no local que preferir. Nesse arquivo, você encontrará uma pasta "cool". Após isso, crie o diretório em que você irá implementar o trabalho. O seguinte comando deve ser executado *dentro* do diretório criado:

```
> make -f caminho/para/cool/assignments/parser/Makefile
```

Este comando irá copiar uma série de arquivos para o seu diretório. Alguns destes serão copiados apenas como leitura (*read-only*). Você **não deve** modificar tais arquivos. Dentre os arquivos copiados, você encontrará um arquivo README com algumas instruções. Os arquivos que você **deve** modificar são:

- cool.y: Este arquivo contém um template inicial de um analisador sintático para *Cool*. A seção de declarações está quase completa, mas você precisará incluir novas declarações de tipos para símbolos não terminais que você venha a introduzir. No caso de símbolos terminais, todas as declarações necessárias já foram adicionadas. Você pode precisar, também, acrescentar declarações de precedência. A seção de regras, no entanto, está praticamente incompleta, com exceção de partes de algumas regras que já foram disponibilizadas. Você não precisa modificar as partes de regras já implementadas, mas você pode se achar necessário. Contudo, não assuma que nenhuma regra em particular está completa.
- good.cl and bad.cl: Estes arquivos testam algumas características da gramática. Você deve adicionar novos testes para garantir que good.cl exercite qualquer construção legal da gramática e que bad.cl exercite a maior quantidade possível de erros. Você deverá explicar os testes que foram adicionar nesses arquivos no arquivo README a ser entregue.
- README: Este arquivo contém, inicialmente, uma lista com os arquivos copiados, além de algumas instruções para compilação e teste do projeto, bem como preparação do arquivo de submissão. Ao final do arquivo, você vai encontrar uma linha com os dizeres **"corte aqui"**. Abaixo desta linha você deve explicar qualquer decisão que tenha tomado quanto ao projeto, os casos de teste adicionados e por que você acredita que seu programa está correto. Antes de preparar o arquivo a ser submetido, certifique-se de que você apagou todo o conteúdo acima do "corte aqui" (incluindo a própria linha).

3 Testando o Parser

Você utilizará uma implementação padrão do analisador léxico (seu trabalho é implementar apenas o analisador sintático). Portanto, confira se existe um executável com o nome lexer no diretório em que foi executado o comando make -f <...>. Caso contrário, execute os seguintes comandos dentro da pasta em que foram copiados os arquivos:

```
> make clean
> make
```

Para testar seu analisador sintático, você utilizará o arquivo de script myparser, que une o analisador léxico ao analisador sintático:

```
> ./myparser foo.cl
```

Note que myparser aceita como argumento uma flag -p para debug; ao utilizar esta flag, serão apresentadas (em stdout) diversas informações sobre o que está sendo feito pelo parser. Além disso, `bison` produz um *dump* (em formato legível) das tabelas de parsing do LALR(1) no arquivo cool.output. Examinar este dump pode ser útil no processo de debug do parser. Por fim, lembre-se que você deve testar tanto programas "bons" quanto programas com erros, para garantir que está tudo funcionando corretamente.

4 Saída do Parser

Suas ações semânticas devem produzir uma AST. A raíz (e somente a raíz) da AST deve ser do tipo `program`. Para programas em que o processo de parsing seja finalizado com sucesso, a saída será uma listagem da AST. Em contraste, para programas que contém erros, a saída será uma mensagem de erro emitida pelo parser. No arquivo cool.y já existe uma rotina de reportação de erros em um formato padrão; por favor, **não a modifique**. Você também **não deve** invocar essa rotina diretamente nas ações semânticas; `bison` a invoca automaticamente quando um problema é detectado. Seu analisador deve funcionar apenas para programas implementados em um único arquivo – não se preocupe com a compilação de múltiplos arquivos.

5 Tratamento de Erros

Você deve usar o (pseudo) não terminal `error` para adicionar recursos de tratamento de erro ao parser. O propósito do não terminal `error` é permitir que o parser continue após algum erro. No entanto, ele não é uma "panacéia" e, por isso, o parser pode ficar completamente confuso. Veja a documentação do `bison` para entender como utilizar `error` da melhor maneira. No seu arquivo de documentação README, descreva os erros que você tentou capturar. Seu arquivo de teste bad.cl deve conter algumas instâncias que ilustrem os erros de que seu parser é capaz de se recuperar. Para receber nota máxima, seu parser deve ser capaz de 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 está sintáticamente correta, então o parser deve ser capaz de reiniciar a partir próxima definição de classe.

- De maneira similar, o parser deve conseguir se recuperar de erros em *features* (movendo para a próxima *feature*), uma declaração em uma expressão `let` (movendo para a próxima variável) e uma expressão dentro de um bloco {...}.

Não se preocupe tanto com o número da linha que aparece nas mensagens de erros geradas pelo parser. Se seu parser está funciona corretamente, o número da linha será, geralmente, a linha em que o erro aconteceu. Para construções incorretas quebradas em várias linhas, o número será, provavelmente, o da útima linha da construção.

6 O Pacote de Árvores

Existe uma discussão extensiva da versão em C++ do pacote de árvores para as ASTs de `Cool` no documento *A Tour of The Cool Support Code* (cool-tour.pdf). Você precisará de boa parte das informações presentes em tal documento para escrever o parser.

7 Observações

- Você pode usar declarações de precedência, mas apenas para expressões. Não utilize declarações de precedência "cegamente" (i.e. não responda a um conflito shift-reduce em sua gramática adicionando regras de precedência até que o conflito deixe de existir).
- A construção `let` de `Cool` introduz ambiguidade a linguagem (tente construir um exemplo, caso não esteja convencido disso). O manual resolve essa ambiguidade ao dizer que uma expressão `let` se extende para a direita tanto quant possível. A ambiguidade aparecerá nos eu parser como um conflito shift-reduce envolvendo as produções para a expressão `let`. Esse problema tem uma solução simple, mas um pouco obscura. Nós não iremos te dizer a solução exata, mas lhe daremos uma dica. Em coolc (versão de referência do compilador de `Cool`), a solução para o conflito shift-reduce em expressões `let` foi implementada através de um recurso do `bison` que permite associar precedência com produções (não apenas operadores). Dê uma olhada na documentação do `bison` para mais informações sobre como utilizar este recurso.
- Atenção: como o compilador mycoolc utiliza *pipes* para comunicação entre uma fase da compilação e a próxima, quaisquer caracteres estranhos produzidos pelo parser podem causar erros; em particular, o analisador semântico pode não conseguir interpretar a AST produzida pelo seu parser.
- Ao executar `bison` no esqueleto inicial do arquivo (cool.y), alguns *warnings* sobre não terminais e regras "inúteis" serão emitidos. Isto é porque alguns dos não terminais e regras não estão sendo utilizados, mas essas mensagens *devem* ir embora assim que seu parser estiver completo.
- Você deve declarar "tipos" de `bison` para os não terminais e terminais que possuem atributos. Por exemplo, no esqueleto inicial do arquivo cool.y existe a seguinte declaração:

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

  Esta declaração diz que o não terminal `program` tem tipo `<program>`. O uso da palavra "tipo" pode ser mal entendido aqui; o que realmente significa é que o atributo do não terminal `program` é armazenado no membro `program` da declaração `union` em cool.y, que tem tipo `Program`. Ao especificar o tipo

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

você está dizendo ao `bison` que os atributos dos não terminais (ou terminais) `X`, `Y` e `Z` têm tipo apropriado para o membro `member_name` do `union`.

Todos os membros da declaração `union` e seus tipos têm nomes similares. É uma coincidência, no exemplo acima, que o não terminal `program` tenha o mesmo nome do membro do `union`. É crucial que você declare os tipos corretos para os atributos dos símbolos da gramática; falhas nesse sentido praticamente garantem que seu parser não irá funcionar. Você não precisa declarar tipos para símbolos da gramática que não possuem atributos.

- O checador de tipos de `g++` emite *warnings* caso você utilize constructores de árvore com parâmetros com tipo errados. Caso você ignore essas mensagens, seu programa pode falhar quando o construtor perceber que está sendo usado incorretamente. Além disso, `bison` pode "reclamar" caso você cause erros de tipo. Preste atenção a todos os avisos. Não se supreenda caso seu programa falhe quando `bison` e `g++` emitem *warnings*.