Trabalho Prático N.02 (Valor: 05 pontos)
Entrega: Conforme Tarefa no Canvas

1 Visão geral

As tarefas de programação – TPs 02 a 05 – irão direcioná-lo para projetar e construir um compilador para Cool. Cada tarefa cobrirá um componente do compilador: análise léxica, análise sintática, análise semântica e geração de código irá resultar em uma fase do compilador completamente funcional que deverá interagir com outras fases. Você terá a opção de fazer seus projetos em C++ ou Java.

Para esta tarefa, você deve implementar um analisador léxico, também chamado de *scanner*, usando um *lexical analyzer generator* – a ferramenta C++ é chamada Flex; a ferramenta Java é chamada JLex. Você descreverá o conjunto de *tokens* para Cool em um formato de entrada apropriado e o gerador de analisador léxico produzirá o código propriamente dito (em C++ ou Java) para reconhecimento das *tokens* em programas escritos em Cool.

A documentação *on-line* para todas as ferramentas necessárias para o projeto foi disponibilizada no **Canvas**. Isto inclui manuais para Flex e JLex (usados nesta tarefa), a documentação para Bison e Java CUP (usados na próxima tarefa), bem como a manual para o simulador de assembly para MIPS denominado spim.

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

2 Introdução ao Flex / JLex

O Flex permite que você implemente um analisador léxico escrevendo regras que correspondam a expressões regulares definidas pelo usuário e executando uma ação específica para cada padrão correspondente encontrado na entrada O Flex traduz um arquivo de regras (por exemplo, "lexer.flex") para o código-fonte em C (ou, se você estiver usando JLex, em Java) implementando um autômato finito que reconhece as expressões regulares que você especificou em seu arquivo de regras. Felizmente, não é necessário entender (ou mesmo, examinar) o arquivo gerado automaticamente (que é, geralmente muito, confuso) que implementa suas regras.

Os arquivos de regras no Flex são estruturados da seguinte maneira:

```
%{
Declarations
%}
Definitions
%%
Rules
%%
User subroutines
```

As seções `Declarations` e `User subroutines` são opcionais e permitem escrever declarações e funções auxiliares em C (ou em Java, no caso do `JLex`). A seção `Definitions` também é opcional, mas geralmente é muito útil, pois as definições permitem atribuir nomes a expressões regulares. Por exemplo, a definição

```
DIGIT [0-9]
```

permite que você defina um dígito. Aqui, `DIGIT` é o nome dado à expressão regular que corresponde a um único caractere qualquer entre 0 e 9.

A tabela a abaixo fornece uma visão geral das expressões regulares comuns que podem ser especificadas no `Flex`:

`x`	o caractere "x"
`‘‘x’’`	um "x", mesmo que x seja um operador
`\x`	um "x", mesmo que x seja um operador
`[xy]`	o caractere x ou y
`[x-z]`	o caractere x, y ou z
`[^x]`	qualquer caractere exceto x
`.`	qualquer caractere exceto mudança de linha
`^x`	um x no início de uma linha
`<y>x`	um x quando a ferramenta Lex está na condição inicial y
`x$`	um x no final de uma linha
`x?`	um x opcional
`x*`	0, 1, 2, ... instâncias of x
`x+`	1, 2, 3, ... instâncias of x
`x|y`	um x ou um y
`(x)`	um x
`x/y`	um x mas somente de for seguido de y
`{xx}`	tradução de xx da seção de definições
`x{m,n}`	de m a n ocorrências de x

A parte mais importante do seu analisador léxico é a seção de regras. Uma regra no `Flex` especifica uma ação a ser executada se a entrada corresponder à expressão ou definição regular no início da regra. A ação a ser executada é especificada escrevendo-se um código-fonte em C/C++ (ou em Java).

Por exemplo, supondo que um dígito represente uma *token* em uma determinada linguagem (note que esse não é o caso da linguagem Cool), a regra:

```
{DIGIT} {
    cool_yylval.symbol = inttable.add_string(yytext);
    return DIGIT_TOKEN;
}
```

registra o valor do dígito na variável global `cool_yylval` e retorna o código do *token* apropriado. Veja Seção 5 para uma discussão mais detalhada da variável global `cool_yylval` e veja a Seção 4.2 para uma discussão sobre o `inttable` usado no fragmento de código acima.

Um ponto importante a ser lembrado é que, se a entrada atual (ou seja, o resultado da chamada de função para `yylex()`) corresponder a várias regras, o `Flex` selecionará a regra que corresponde ao maior número de caracteres. Por exemplo, se você definir as duas regras como as seguintes:

```
[0-9]+      { // ação 1}
[0-9a-z]+   { // ação 2}
```

e caso a sequência de caracteres `‘2a’` aparecer em seguida no arquivo que está sendo processado, então a `ação 2` será executada, já que a segunda regra corresponde a mais caracteres que a primeira regra. Se várias regras corresponderem ao mesmo número de caracteres, a regra que aparecer primeiro no arquivo será escolhida.

Ao escrever regras no `Flex`, pode ser necessário executar ações diferentes, dependendo das *tokens* encontradas anteriormente. Por exemplo, ao processar uma *token* de fechamento de comentário, você pode estar interessado em saber se uma *token* de abertura de comentário foi encontrada anteriormente.

Uma maneira óbvia de rastrear o estado é declarar variáveis globais em sua seção de declaração, que são configuradas como verdadeiras quando determinadas *tokens* de interesse são encontradas. Além disso, o `Flex` também fornece uma "facilidade sintática" para se obter uma funcionalidade semelhante por meio de declarações de estado, como por exemplo:

```
%Start COMMENT
```

que pode ser definida como verdadeira por meio da instrução `BEGIN(COMMENT)`. Para se executar uma ação somente se uma abertura de comentário foi encontrada anteriormente, você pode condicionar sua regra usando `COMMENT` e a seguinte sintaxe:

```
<COMMENT> {
   // o restante do código de sua regra ...
}
```

Há também um estado padrão especial chamado `INITIAL`, que está ativo, a menos que você indique explicitamente o início de um novo estado. Você pode achar essa construção útil para vários aspectos desse trabalho, como por exemplo retorno de erros.

Sinta-se encorajado a ler a documentação sobre `Flex` disponibilizada no **Canvas**, antes de iniciar a escrita de seu próprio analisador léxico. Da mesma forma, recomenda-se usar o livro a seguir como referência para estudos de `Flex` e `Bison`:

LEVINE, John. Flex & Bison: Text Processing Tools. " O'Reilly Media, Inc.", 2009.

3 Arquivos e Diretórios

Para começar, crie um diretório onde você deseja fazer o desenvolvimento de seu trabalho (por exemplo, `PA2`) e execute um dos seguintes comandos *nesse diretório*. Para a versão em C++ do trabalho, você deve digitar:

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

Enquanto que para a versão em Java, digite:

```
cd PA2
make -f /var/tmp/cool/assignments/PA2J/Makefile
```

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

Este comando irá copiar um número de arquivos para o seu diretório. Alguns dos arquivos serão copiados somente para 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.flex (na versão em C++) / cool.lex (na versão em Java)
  Este arquivo contém um esqueleto de uma descrição léxica para a liguagem Cool. Há comentários indicando onde você precisa preencher o código, mas isso não é necessariamente um guia completo. Parte deste trabalho é certificar-se de que você tem um *scanner* correto e funcional. Exceto pelas seções indicadas em contrário, você é bem-vindo para fazer modificações nesse esqueleto. Você já pode realmente construir um *scanner* com a descrição do esqueleto, mas ele não faz muito. Você deve ler o manual Flex/Jlex para descobrir o que esta descrição faz. Quaisquer rotinas auxiliares que você deseja escrever devem ser adicionadas diretamente a este arquivo na seção apropriada (veja os comentários no arquivo).

- test.cl
  Este arquivo contém uma entrada de exemplo a ser processada. Ele não cobre toda a especificação léxica da linguagem Cool; mas representa, no entanto, um teste interessante. **Não é um bom teste para se começar, nem fornece testes adequados para a versão final do seu *scanner*.** Parte de sua tarefa é apresentar boas entradas de teste e uma estratégia de teste (que deverá ser descrita nas anotações sobre a implementação). Não menospreze essa parte da tarefa — é difícil criar uma boa entrada para teste e o esquecimento/falta de teste por algo é a causa mais provável da perda de pontos durante a avaliação desse trabalho.

  Você deve modificar este arquivo com testes que você acha que verifiquem adequadamente seu *scanner*. A versão do arquivo test.cl fornecida é semelhante a um programa "real" na linguagem Cool, porém seus testes não precisam ser. Além disso, você pode manter a porção que julgar adequada dessa versão de teste disponibilizada previamente.

- README
  Este arquivo contém instruções detalhadas para a tarefa, além de várias dicas úteis. Você também deve editar esse arquivo para incluir anotações sobre o seu projeto. Você deve explicar as decisões de design, por que sua implementação está correta e por que seus casos de teste são adequados. Faz parte da avaliação dessa tarefa fornecer uma explicação de forma clara e concisa, bem como comentar sua implementação.

Embora inicialmente esses arquivos estejam incompletos, é possível compilar e executar uma solução provisória que apenas lista o nome do arquivo e seu conteúdo (basta usar, make lexer).

4 Resultados do *Scanner*

Você deve seguir a especificação da estrutura lexical do Cool fornecida na Seção 10 e na Figura 1 do *Manual de Referência da Linguagem* Cool. Seu *scanner* deve ser robusto – isto é, deve funcionar para qualquer entrada concebível. Por exemplo, você deve manipular erros como um `EOF` ocorrendo no meio de um string ou comentário, bem como constantes do tipo string que são muito longas. Estes são apenas alguns dos erros que podem ocorrer; veja o manual para o resto.

Você deve se precaver para realizar uma finalização normal quando ocorrer um erro fatal. *Core dumps* ou exceções não detectadas são inaceitáveis!

4.1 Tratamento de Erros

Todos os erros devem ser passados para o *parser*. Seu analisador léxico não deve imprimir nada. Erros são comunicados ao *parser* retornando uma *token* especial de erro chamada **ERROR**. Note que você deve ignorar a *token* chamada **error** [em letras minúsculas] para esta tarefa; ela será usada pelo parser no TP03.

Existem vários requisitos para relatar e recuperar de erros léxicos:

- Quando um caractere inválido (um que não pode iniciar nenhuma *token*) for encontrado, um string contendo apenas esse caractere deve ser retornado como string de erro. Retomar a análise léxica no caractere seguinte.

  Se um string contiver uma nova linha sem caractere de *escape*, deve-se reportar esse erro como `"Unterminated string constant"` e retomar a análise léxica no início da próxima linha – na suposição de que o programador simplesmente esqueceu de fechar/terminar o string.

- Quando um string é muito longo, relatar o erro como `"String constant too long"` no string de erro do token **ERROR**. Se o string contiver caracteres inválidos (isto é, o caractere nulo), relatar isso como

‘‘`String contains null character`’’. Em ambos os casos, a análise léxica deve continuar após o final do string, sendo o final do string definido como

1. o início da próxima linha se uma nova linha sem caractere de *escape* ocorrer após esses erros serem encontrados; ou
2. após as aspas (”) de fechamento, caso contrário.

- Se um comentário permanecer aberto quando o `EOF` for encontrado, relatar este erro com a mensagem ‘‘`EOF in comment`’’. **Não se deve “*tokenizar*” o conteúdo do comentário simplesmente porque o terminador está faltando.** De forma análoga para os strings, se um `EOF` for encontrado antes de seu fechamento, relatar este erro como ‘‘`EOF in string constant`’’.
- Se você encontrar “`*)`” fora de um comentário, relatar este erro como ‘‘`Unmatched *)`’’, ao invés de representá-lo como `*` e `)`.

4.2 Tabela de Strings

Programas tendem a ter muitas ocorrências do mesmo lexema. Por exemplo, um identificador é geralmente chamado mais de uma vez em um programa (ou então não é muito útil!).

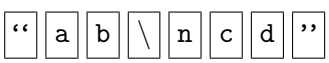
Para economizar espaço e tempo, uma prática comum de compilador é armazenar lexemas em uma *tabela de strings*. Será fornecida uma implementação de tabela de strings tanto para a versão em C++ como para a versão em Java. Veja as seções a seguir para detalhes.

Há uma questão importante envolvendo a decisão sobre como lidar com os identificadores especiais para as classes básicas (**Object**, **Int**, **Bool**, **String**), `SELF_TYPE`, e `self`. No entanto, esse problema não surge até as fases posteriores do compilador — o seu *scanner* deve tratar os identificadores especiais exatamente como qualquer outro identificador.

Não se deve testar se literais inteiros se encaixam na representação especificada no manual Cool — simplesmente crie um símbolo com o texto do literal inteiro como seu conteúdo, independentemente de seu comprimento.

4.3 Strings

Seu *scanner* deve converter caracteres de *escape* nas constantes de string em seus valores corretos. Por exemplo, se o programador digitar esses oito caracteres:



seu *scanner* retornaria o token **STR_CONST** cujo valor semântico é esses 5 caracteres:

em que  representa o caractere ASCII para uma nova linha.

Seguindo a especificação na página 15 do manual Cool, você deve retornar um erro para um string contendo o caractere nulo. No entanto, a sequência de dois caracteres

é permitida, mas deve ser convertida para o caractere

.

4.4 Outras Observações

Seu *scanner* deve manter a variável **curr_lineno** que indica qual linha no texto/arquivo de origem está sendo processada no momento. Esse recurso ajudará o *parser* na exibição de mensagens de erro úteis.

Você deve ignorar a token **LET_STMT**. Ela será usada apenas pelo parser (no TP03). Por fim, observe que, se a especificação léxica estiver incompleta (determinadas entradas não possuirem nenhuma expressão regular que as corresponda), os *scanners* gerados por Flex e Jlex fazem coisas indesejáveis. **Certifique-se de que sua especificação esteja completa**.

5 Notas adicionais para a versão em C++

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

- Cada chamada do *scanner* retorna a próxima *token* e o lexema de entrada. O valor retornado pela função **cool_yylex** é um código inteiro representando a categoria sintática (por exemplo, literal inteiro, ponto-e-vírgula, palavra-chave `if`, etc.). Os códigos para todas as *tokens* são definidos no arquivo cool-parse.h. O segundo componente, o valor semântico ou lexema, é colocado na união (declarada de forma global) **cool_yylval**, que é do tipo `YYSTYPE`. O tipo `YYSTYPE` também é definido em cool-parse.h. As tokens para símbolos de caractere único (por exemplo, ";" e ",") são representados apenas pelo valor inteiro (ASCII) do próprio caractere. Todas as *tokens* de caracteres únicos estão listados na gramática da linguagem Cool no manual disponibilizado.

- Para identificadores de classe, identificadores de objeto, inteiros e strings, o valor semântico deve ser um valor do tipo **Symbol** (ver item seguinte) armazenado no campo **cool_yylval.symbol**. Para constantes booleanas, o valor semântico é armazenado no campo **cool_yylval.boolean**. Exceto para erros (veja mais adiante), os lexemas das outras *tokens* não contêm nenhuma informação interessante.

- É fornecida uma implementação de tabela de strings, que é discutida em detalhes na *Descrição do Código de Suporte para Implementação da Linguagem COOL* (ou em inglês, *A Tour do Cool Support Code*) e na documentação do código. Por enquanto, você só precisa saber que o tipo das entradas da tabela de strings é **Symbol**, pois ela deverá ser usada para armazenar o valor semântico (lexema) associado com identificadores de classe, identificadores de objeto, inteiros e strings.

- Quando um erro léxico é encontrado, a rotina **cool_yylex** deve retornar a token **ERROR**. O valor semântico é um string que representa a mensagem de erro, que é armazenada no campo **cool_yylval.error_msg** (observe que esse campo é um string comum, **não um símbolo**). Consulte a seção anterior para obter informações sobre o que colocar em mensagens de erro.

6 Notas adicionais para a versão em Java

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

- Cada chamada do *scanner* retorna a próxima *token* e o lexema da entrada. O valor retornado pelo método **CoolLexer.next_token** é um objeto da classe **java_cup.runtime.Symbol**. Este objeto tem um campo representando a categoria sintática de uma *token* (por exemplo, literal de inteiro, ponto-e-vírgula, a palavra-chave `if`, etc.). Os códigos sintáticos para todos as *tokens* são definidos no arquivo TokenConstants.java. O componente, o valor semântico ou lexema (se houver), também é colocado em um objeto **java_cup.runtime.Symbol**. A documentação da classe **java_cup.runtime.Symbol**, bem como outro código de suporte, foi disponibilizado via **Canvas**. Exemplos de seu uso também estão disponíveis.

- Para identificadores de classe, identificadores de objeto, inteiros e strings, o valor semântico deve ser do tipo **AbstractSymbol** (ver item seguinte). Para constantes booleanas, o valor semântico é do tipo **java.lang.Boolean**. Exceto para erros (veja mais adiante), os lexemas das outras *tokens* não contêm nenhuma informação interessante. Como o campo **value** da classe **java_cup.runtime.Symbol** tem tipo genérico **java.lang.Object**, você precisará convertê-lo em um tipo apropriado antes de chamar qualquer método sobre nele.
- É fornecida uma implementação de tabela de strings, que é definida em AbstractTable.java. Por enquanto, você só precisa saber que o tipo das entradas da tabela de strings é **AbstractSymbol**, pois ela deverá ser usada para armazenar o valor semântico (lexema) associado com identificadores de classe, identificadores de objeto, inteiros e strings..
- Quando um erro léxico é encontrado, a rotina **CoolLexer.next_token** deve retornar um objeto **java_cup.runtime.Symbol** cuja categoria sintática é **TokenConstants.ERROR** e cujo valor semântico é o string de mensagem de erro. Veja Seção 4 para informações sobre como construir mensagens de erro.

7 Testando o *Scanner*

Existem pelo menos duas maneiras de testar seu *scanner*. A primeira maneira é gerar "entradas-exemplos" e analisá-las usando lexer, que imprime o número da linha e o lexema de cada *token* reconhecida pelo seu *scanner*.

Uma outra maneira de realizar testes, quando você acreditar que seu *scanner* está funcionando corretamente, é tentar executar mycoolc para invocar seu analisador léxico junto com todas as outras fases do compilador (que serão fornecidas). Neste caso, mycoolc funcionará como um compilador completo para a linguagem Cool que você pode experimentar sobre qualquer programa de teste.

8 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 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 `PA2`, por exemplo da seguinte forma:

```
tar cvzf PA2.tar.gz PA2
uuencode PA2.tar.gz PA2.tar.gz > PA2.u
rm PA2.tar.gz
```

4. Entregar o arquivo `PA2.u` via **Canvas**.