Trabalho Prático 2
Sexta, Outubro 7, 2016

1 Introdução

Os trabalhos práticos de 2 a 5 irão te direcionar no projeto e implementação de um compilador para linguagem Cool. Cada trabalho irá cobrir um componente do compilador: análise léxica, análise sintática, análise semântica e geração de código. Cada trabalho será uma fase do compilador que comunica com as outras fases.

Para esse trabalho, você irá escrever o analisador léxico, também chamado de *scanner*, utilizando um *gerador de analisador léxico* chamado flex. Você irá descrever o conjunto de tokens para Cool em um formato apropriado e o gerador de analisador irá gerar um código em C++ para reconhecer os tokens de programas escritos em Cool.

Documentação para as ferramentas necessárias para o projeto estarão disponíveis. Incluindo o manual do flex (usado nesse trabalho), documentação do bison (usado no próximo trabalho) e, por último, o manual do simulador spim.

2 Arquivos e Diretórios

Para começar, no seu diretório home, em alguma máquina do laboratório da graduação (máquinas *.grad) crie um diretório onde irá implementar o seu trabalho e execute o comando abaixo:

```
make -f /home/prof/renato/cool/student/assignments/PA2/Makefile
```

O trabalho será avaliado em uma dessas máquinas Linux portanto utilize umas delas.

Este comando irá copiar alguns arquivos para o seu diretório. Alguns desses arquivos serão copiados como "read-only" (utilizando links simbólicos pro arquivo original). Você não deve modificar esses arquivos. Caso você faça uma cópia desses arquivos pode ser que você não consiga implementar o trabalho. Leia as instruções do README. Os arquivos que você precisa modificar são:

- cool.flex
  Este arquivo contém um esqueleto para uma descrição léxica de Cool. Existem comentários indicando onde você precisa preencher código mas não é um guia completo. Parte do trabalho é você garantir que tem um analisador léxico correto e funcionando. Exceto pela seções indicadas, fique livre para fazer modificações no nosso esqueleto. Você pode implementar um scanner com a nossa descrição mas ele não faz muita coisa. Você deve ler o manual do flex para descobrir o que a nossa descrição faz. Qualquer rotina auxiliar que você deseja escrever deve ser adicionada nesse arquivo na seção apropriada (veja os comentários no arquivo)

- test.cl
  Este arquivo contém alguns exemplos de entrada pra teste. Esse teste não atende toda a especificação mas é um teste interessante. Não é um bom teste para começar e nem é um teste adequado. Parte do trabalho é escrever bons testes e boas estratégias. (Não pegue leve, bons testes são complicados de criar e esquecer de testar alguma coisa provavelmente irá prejudicar a sua nota).

Você deve modificar esse arquivo com testes que você julgar adequado ao seu scanner. Nosso test.cl é parecido com um programa real em Cool mas os seus testes não necessariamente precisam ser. Você é livre para definir o volume de testes, muito ou pouco.

- README
  Este arquivo contém instruções detalhadas para o trabalho e algumas dicas. Você pode editar esse arquivo caso queira comentar alguma coisa do seu projeto. Você pode escrever decisões de implementação, explicar porque seu código está correto e porque seus testes são adequados. É interessante que você tenha uma explicação clara e objetiva do seu trabalho assim como comentar seu código.

Apesar dos arquivos estarem incompletos inicialmente o analisador léxico compila e executa (make lexer).

3 Resultados

Você pode seguir a especificação da estrutura léxica de Cool de acordo com a seção 10 e Figura 1 do manual do Cool. Seu scanner deve ser robusto—deve funcionar para qualquer entrada concebível. Por exemplo, você deve tratar erros como caracter EOF no meio de uma string ou comentário, assim como também deve tratar strings que são muito grandes. Esses são apenas alguns erros que podem ocorrer, veja o manual para o resto.

O scanner deve terminar natualmente mesmo se um erro fatal ocorrer. Core dumps, stack trace, exceções não tratadas e etc não devem ocorrer.

3.1 Tratamento de Erro

Todos erros devem ser enviados ao analisador sintático (parser). Seu analisador léxico (lexer) não deve imprimir nada. Erros são comunicados ao parser retornando um caracter especial chamado **error**. (Observação, você deve ignorar o token chamado **error** [em minúsculo] para este trabalho; será utilizado apenas no próximo trabalho. Haverão vários requisitos para sinalizar e recuperar de erros léxicos:

- Quando um caracter inválido (um que não pode começar com nenhum token) é encontrado, uma string contendo somente este caracter deve ser retornada como a string de erro. Continue a análise léxica pelo próximo caracter.
- Se uma string contem um caracter de nova linha não escapado sinalize o erro como `‘‘Unterminated string constant’’` e continue a análise na próxima linha—assumiremos que o programador simplesmente esqueceu de escapar
- Quando uma string é muito longa, sinalize o erro como `‘‘String constant too long’’` na string de erro no token **ERROR**. Se a string contém caracteres inválidos (por exemplo o caracter null), sinalize como `‘‘String contains null character’’`. Em ambos os casos, a análise léxica deve continuar após o final da string. O final da string é definido como uma das opções abaixo:
  1. o começo de uma nova linha se uma nova linha não escapada ocorrer após esses erros forem encontrados; ou
  2. após a string ser fechada com ”

- Se um comentário permanece aberto quando EOF é encontrado sinalize esse erro com a mensagem ``EOF in comment''. Faça a mesma coisa para strings, se um EOF for encontrado antes de fechar a string sinalize o erro como ``EOF in string constant''.
- Se você encontrar "*)" fora de um comentário, sinalize esse erro como ``Unmatched *)'' ao invés de gerar tokens * e).

3.2 Tabela de String

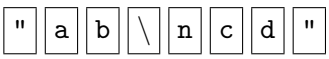
Programas tendem a ter várias ocorrências do mesmo lexema. Por exemplo, um identificador é geralmente referenciado mais de uma vez em um programa (do contrário ele não seria muito útil!). Para economizar espaço e tempo, uma boa prática é salvar os lexemas em uma *Tabela de String*. Nós disponibilizamos uma tabela de símbolo em C++.

Há um problema na decisão de como manipular os identificadores especiais para as classes básicas (**Object**, **Int**, **Bool**, **String**), SELF_TYPE, e self. Entretanto, esse problema não irá aparecer por equanto, somente nas últimas fases do compilador—o scanner deve tratar esses identificadores especiais exatamente como outros identificadores.

Não teste se a representação de um inteiro corresponde a representação especificada no Manual do Cool—simplesmente crie um Símbolo com todo o conteúdo textual, independente do seu tamanho.

3.3 Strings

Seu scanner deve converter caracteres escapados em strings constantes para seus valores esperados. Por exemplo, se o programador digitar esses oito caracteres:



seu scanner retornará o token **STR_CONST** no qual o valor semântico são os seguintes 5 caracteres:

onde \n representa o caracter literal ASCII para quebra de linha.

Seguindo a especificação na página 15 do manual do Cool, você deve retornar um erro para uma string contendo o caracter literal null. Contudo, a sequência de dois caracteres:

é permitida mas deve ser convertida para o caracter único

.

3.4 Outras Notas

Seu scanner deve manter a variável **curr_lineno** que indica qual linha no código fonte está sendo lida no momento. Essa funcionalidade irá auxiliar o parser a imprimir mensagens de erros úteis.

Você deve ignorar o token **LET_STMT**. Ele será utilizado apenas pelo parser (TP3). Finalmente, caso a sua especificação léxica estiver incompleta (algumas entradas não possuem expressões regulares correspondentes), então o seu scanner gerado pelo flex irá ter comportamentos indesejados. *Tenha certeza que a sua especificação está completa*

4 Algumas instruções

- *Cada chamada no scanner retorna o próximo token e lexema vindo da entrada. O valor retornado pela função* **cool_yylex** *é um inteiro que representa uma categoria sintática (por exemplo: inteiro literal, ponto e vírgula, palavra reservada* `if`*, etc.). O código para todos os tokens estão definidos no arquivo* cool-parse.h*. O segundo componente, o lexema ou valor semântico, é colocado dentro da union* **cool_yylval***, que é do tipo YYSTYPE. O tipo YYSTYPE também está definido em* cool-parse.h*. Os tokens para caracteres individuais (";" e ",") são representados somente pelo valor inteiro (ASCII) do próprio caracter. Todos os tokens de caracteres individuais estão listados na gramática do Cool no manual do Cool.*

- *Para identificadores de classe, identificadores de objetos, inteiros, e strings, o valor semântico deve ser um* **Symbol** *armazenado no campo* **cool_yylval.symbol***. Para constantes binárias, o valor semântico é armazenado no campo* **cool_yylval.boolean***. Exceto pelos erros (veja abaixo), os lexemas para os outros tokens não carregam nenhuma informação interessante.*

- *Nós disponibilizamos para você uma implementação da Tabela de String, que é discutida em detalhes em* A Tour of the Cool Support Code *e na documentação do código. Por enquanto, você só precisa saber que o tipo das entradas da Tabela de String é* **Symbol***.*

- *Quando um erro léxico é encontrado, a rotina* **cool_yylex** *deve retornar o token* **ERROR***. O valor semântico é a string que representa a mensagem de erro, que é armazenada no campo* **cool_yylval.error_msg** *(perceba que esse campo é uma string qualquer, não um símbolo). Veja a sessão anterior para saber o que colocar nas mensagens de erro.*

5 Testando o Scanner

Há pelo menos dois jeitos de você testar o seu scanner. O primeiro é gerar entradas de exemplo e executá-las usando lexer*, que irá exibir o número da linha e o lexema de cada token reconhecido pelo seu scanner. O outro jeito, quando você julgar que seu scanner está funcionando, é tentar executar mycoolc para executar seu lexer juntamente com as outras fases do compilador (que iremos disponibilizar). Isso será um compilador Cool completo que você poderá executar com qualquer programa de teste.*