EA872 - Laboratório de Programação de Software Básico

Atividade 3

1. Tema

Interpretadores de comandos, compiladores e ferramentas para geração de novas linguagens.

2. Objetivos

Familiarização com processos de geração automática de analisadores léxicos e sintáticos. Compreensão dos papéis complementares entre a análise léxica e a sintática no processo de reconhecimento de uma sentença válida. Uso de gramáticas formais para especificação das funcionalidades de analisadores. Associação de ações semânticas (síntese de resultados) aos comandos interpretados.

3. Atividades

Esta atividade é uma continuação da Atividade 2 realizada na última semana. Nela serão exercitados conceitos mais avançados de analisadores léxicos e sintáticos com a criação e avaliação destes dois analisadores atuando em conjunto. O aluno deverá rever o roteiro da Atividade 2 e o tutorial sobre Lex e Yacc indicado por um link na página da disciplina, pois parte do conteúdo necessário aqui está presente nestes documentos.

3.1. Atividades para entrega ao final da aula

(ex1) (1 pt.) Considere a especificação lex abaixo.

```
%{
    int c, p, l;
%}
%%
\n          { l++; c++; }
[^ \t\n]+   { p++, c += yyleng; }
.           { c++; }
%%
int main(void) {
    yylex();
    printf("c=%d\tp=%d\tl=%d\n",c,p,l);
    return 0;
}
```

(ex1.a) Qual é a finalidade desta especificação?

(ex1.b) Mostre um exemplo de uso do analisador resultante?

(ex1.c) O que representa a variável yyleng e para que ela está sendo usada?

(ex1.d) O que significa o padrão " `[^ \t\n]+` "?

(ex1.e) Por que é necessária a linha " `.  { c++; }` "?

(ex2) (1 pt.) Vejamos agora o funcionamento de um analisador que conjuga as funcionalidades de análise sintática e análise léxica. Copie o código e compile as duas partes, juntando em seguida tudo em um analisador único. Execute o mesmo, veja seu comportamente para diferentes valores de entrada e responda às seguintes questões.

(ex2.a) Qual é a finalidade da linha `#include "ex2.tab.h"` em ex2.l?

(ex2.b) Qual é o propósito das variáveis pré-definidas `yytext` e `yylval`?

(ex2.c) Qual é a função dos parâmetros $$, $1 e $2 na especificação ex2.y?

(ex2.d) Explique qual é o caminho seguido pelo valor digitado na entrada até ser impresso pelo comando

```
printf("Sintático: linha vale (produção composta) = %d\n", $2);
```

(ex2.e) Explique por que é necessária a linha `\n return('\n');` e o que acontece internamente no analisador se a mesma não está presente.

Especificação ex2.l

```
%{
        #include "ex2.tab.h"   // Este include file é gerado pelo bison. Por isso,
                               // compile o analisador sintático antes do léxico.
%}
%%
[0-9a-fA-F]+ { sscanf(yytext, "%x", &yylval);
               printf("Léxico: yytext (texto) = %s \t yyval (valor) =  %d\n", yytext, yylval);
               return (INTEIRO);
             }
\n             return ('\n');
%%
```

Especificação ex2.y

```
%token INTEIRO

%{
        #include <stdio.h>
%}

%%
linhas:   linhas linha {printf("Sintático: linha vale (produção composta) = %d\n", $2); }
        | linha        {printf("Sintático: linha vale (produção simples) = %d\n", $1); }
linha :  INTEIRO '\n'  {printf("Sintático: INTEIRO vale = %d\n", $1);
                        $$ = $1; }
%%

void main()     { yyparse();
                }
```

(ex3) (1 pt.) Vamos evoluir o exemplo anterior para algo mais completo e que consiga fazer algum trabalho útil. Estude, compile e execute as especificações a seguir e responda.

(ex3.a) Detalhe os padrões que são reconhecidos (que casam) com a expressão `[0-9]*\.[0-9]*` .

(ex3.b) A expressão do item anterior possui uma falha que precisa ser corrigida. Que falha é esta e como deve ser corrigida para que não afete mais o correto funcionamento do analisador?

(ex3.c) Faça modificações e teste as especificações para que elas suportem números hexadecimais e mostre os resultados de alguns testes. Documente as linhas que foram alteradas no relatório.

Especificação ex3.l

```
%{

#include "ex3.tab.h"

%}

digitos   [0-9]+

frac      [0-9]*\.[0-9]*

operador  [-+=]

nl        \n

%%

{digitos}      {sscanf(yytext, "%d", &yylval.valor_inteiro);

                return INTEIRO;

               }

{frac}         {sscanf(yytext, "%f", &yylval.valor_real);

                return REAL;

               }

{operador}     return yytext[0];

{nl}           return ('\n');
```

Especificação ex3.y

```
%{
#include <stdio.h>
float resultado = 0;
%}
%union {
        float  valor_real;
        int    valor_inteiro;
       }
%token <valor_inteiro> INTEIRO
%token <valor_real> REAL
%%
linhas:    linha
         | linhas linha
linha:     '\n'        {printf("=%f\n", resultado); }
         | exp '\n'    {printf("=%f\n", resultado); }
exp   : INTEIRO        { resultado += $1; }
        | '+' INTEIRO  { resultado += $2; }
        | '-' INTEIRO  { resultado -= $2; }
        | '=' INTEIRO  { resultado  = $2; }
        | '='          { resultado  =  0; }
        | REAL         { resultado += $1; }
        | '+' REAL     { resultado += $2; }
        | '-' REAL     { resultado -= $2; }
        | '=' REAL     { resultado  = $2; }
%%
void main(){
    yyparse();
}
```

3.2. Atividades para entrega até a próxima aula

(ex4) (3 pts.) Parser para alertar sobre variáveis não iniciadas em C

Crie um parser, usando lex e yacc, que receba como entrada um programa-fonte em uma linguagem C simplificada e emita alertas (warnings) indicando as variáveis que foram usadas (acessadas) sem terem sido iniciadas antes. Uma variável é dita iniciada quando a ela é atribuído algum conteúdo, ou seja, ela aparece à esquerda de um sinal de atribuição "=". Considere que o programa só usa variáveis em linhas de declaração de tipos e em expressões (C simplificado). Para testar o analisador produzido, estão sendo disponibilizados diversos arquivos em C simplificado em um arquivo zip. Seu analisador deverá conseguir gerar os alertas para todos estes programas. Os analisadores léxico e sintático devem ter apenas as regras necessárias para realizar o teste em todos os 10 arquivos de teste que foram disponibilizados, não sendo necessário criar analisadores capazes de avaliar um código em C completo com funções arbitrárias, loops, cases etc.

Dica: à medida que for encontrando sentenças que iniciam o conteúdo das variáveis, o programa pode guardar o nome das mesmas em um **vetor (ou lista) de variáveis liberadas para uso (acesso)** pelo restante do programa.

Obs: para auxiliar no entendimento do que é a definição da gramática de uma linguagem, disponibilizamos abaixo um exemplo parcial de gramática de uma linguagem C simplificada, como a que é usada nos arquivos de teste do exercício ex4. Observem que este arquivo é só uma parte da definição da gramática que deverá ser gravada em um arquivo com extensão .y para alimentar o gerador de analisadores sintáticos yacc ou bison.

```
/* Exemplo de gramática parcial para C simplificado                */
/*                                                                 */
/* Este código não está completo e é só um exemplo inicial.        */
/*                                                                 */
/* Palavras em maiúsculas são TOKENS obtidos do analisador léxico  */
/* e as minúsculas são símbolos internos do analisador sintático.  */
/*                                                                 */
/* EA872 - 2s/2015 - FEEC - Unicamp                                */
%%
prog_fonte     :       INICIO_MAIN conteudo_prog FIM_MAIN
               ;
conteudo_prog  :       declaracoes expressoes
               ;
declaracoes    :       linha_declara declaracoes
               |       linha_declara
               ;
linha_declara  :       TIPO variaveis PONTO_VIRGULA
               ;
variaveis      :       identificador VIRGULA variaveis
               |       identificador
               ;
identificador  :       NOMEVAR
               |       NOMEVAR IGUAL VALOR    /* lembre-se que o conteúdo do TOKEN NOMEVAR */
               ;                              /* é acessível via $1 e pode ser inserido na */
                                              /* lista neste ponto                         */
expressoes     :       linha_executavel expressoes
               |       linha_executavel
               ;
linha_executavel:      NOMEVAR IGUAL operacoes PONTO_VIRGULA /* idem */
               ;
operacoes      :       operacoes OPERA operacoes
               |       PARENTESES_ESQ operacoes PARENTESES_DIR
               |       VALOR
               |       NOMEVAR               /* atenção: variável apareceu aqui novamente */
               ;
%%
```

(ex5) (4 pts) Geração de parser para desmembramento de comandos e armazenamento de informações

O objetivo desta atividade é construir um *parser* utilizando as ferramentas **lex** (ou **flex**) e **yacc** (ou **bison**) com a finalidade de separar informações como comandos e opções/parâmetros para posterior interpretação e utilização por outros programas**.** Considere que a entrada do parser será um arquivo contendo várias linhas de texto separadas em dois campos cada uma: o primeiro campo é formado pelas informações que vem antes do símbolo " : " (dois pontos) representando comandos e o segundo campo é tudo que vem após os dois pontos (são os parâmetros/opções). No campo de comando deve haver um ou mais comandos compostos por qualquer combinação de caracteres que não inclua o sinal de dois pontos. O campo de parâmetros/opções terá um ou mais valores separados por vírgula. Todas as linhas do arquivo são desta forma, podendo haver linhas com o campo de parâmetros vazio. Se o campo de comando estiver vazio, o parser deverá gerar um erro e continuar seu trabalho nas linhas seguintes. Se uma linha for iniciada por um símbolo # ela será considerada comentário e ignorada.

Exemplo de arquivo de entrada:

```
# Arquivo de comandos e parâmetros para o exercício ex5.
comando_a:parâmetro_a1,parâmetro_a2,opção_a1, 324234, x3294549
comando_b : opção_b1, opção_b2, parametro_xyz
comando_c :
: parâmetro_inócuo_1, parâmetro_inútil_1
comando_d:     parâmetro_d, 22, 1, opção_inexistente
```

O parser, após separar o que é comando (primeiro campo) e o que são parâmetros, deverá armazenar tudo em listas ligadas da seguinte forma: deverá haver uma lista chamada **Comandos**, na qual deverão ser inseridos os comandos encontrados. Cada nó desta lista deverá conter um campo para guardar o comando, outro para apontar para o próximo elemento da lista ligada e um outro campo que aponta para o início de uma outra lista. Esta outra lista deverá conter um nó para cada parâmetro e/ou opção daquele comando. Ou seja, de cada nó da lista principal (**Comandos**) sairá uma outra lista contendo um nó para cada parâmetro/opção da linha correspondente. Se não houver parâmetros/opções em um comando, sua lista secundária será uma lista vazia.