AULA PRÁTICA 1

Datas Importantes

- 24/08 - 5 pontos
- 25/08 - 4 pontos
- 26/03 - 3 pontos
- 27/03 - 2 pontos

Entrega

- Mandar email para: benevid@unemat.br
- Colocar no assunto do email: LFA2019-2
- Corpo do Email: Descrever como resolveu o exercício e o que foi alterado.
- Anexar o código fonte.

Gramáticas

Muitas *linguagens formais*, como as linguagens de programação (C, Python, Java, etc.), podem ser descritas através de *gramáticas*, um conjunto de regras recursivas. Com estas notas você vai:

- aprender alguns conceitos fundamentais de gramáticas;
- ver dois exemplos importantes: seqüências balanceadas de parênteses e expressões aritméticas;
- ver como é possível criar um *parser* ou *analisador sintático*, um programa capaz de analisar um texto e descobrir se e como ele pode ser gerado pelas regras de uma dada gramática.

Seqüências balanceadas de parênteses

Uma ***seqüência de parênteses*** é uma string composta unicamente por `(` e `)`. A conceito de seqüência de parênteses ***balanceada*** pode ser definido de forma recursiva:

1. A seqüência vazia é balanceada.
2. Se `S` e `T` são seqüências balanceadas, então `(S)T` é balanceada. (Por exemplo, as seqüências `(()())` e `()()` são balanceadas, e portanto `(((()()))()()` é balanceada.)

As regras acima definem a ***linguagem*** (conjunto de strings) das seqüências de parênteses balanceadas.

Uma maneira formal de definir a linguagem das seqüências balanceadas é através de ***gramáticas livres de contexto***(***context-free grammars***). Podemos fazê-lo através da gramática composta das seguintes ***produções*** ou ***regras***, com ***símbolo inicial*** `B`:

```
1 B = "(" , B , ")" , B ;
2 B = "" ;
```

Acima temos duas regras ou produções. À esquerda de cada regra, antes do sinal de igual `=`, temos um símbolo ***não-terminal***. No caso acima, temos apenas um símbolo não-terminal, `B`. Cada símbolo não-terminal pode ser substituído pela string à direita do sinal de igual. Por exemplo, a primeira produção acima diz que podemos substituir um símbolo `B` pela concatenação de `(`, com um símbolo `B`, com `)` e depois com outro símbolo `B`. (A vírgula representa concatenação; trechos entre aspas duplas são strings chamadas de ***terminais***; o ponto-e-vírgula denota o fim de uma produção.) A segunda produção diz que um símbolo `B` pode ser substituído por uma string vazia `""`.

Para formar uma palavra usando as regras acima, fazemos o seguinte. Começamos com um determinado símbolo não-terminal chamado de ***símbolo inicial***, em nosso caso o `B`. A cada passo substituímos um símbolo não-terminal pelo lado direito de uma de suas produções. No final, devemos obter uma string contendo apenas terminais. Por exemplo, a seqüência balanceada `(()())()` pode ser assim obtida:

```
B -> (B)B -> ((B)B)B -> (()B)B -> (()(B)B)B ->
(()()B)B -> (()())B
  -> (()())(B)B -> (()())()B -> (()())()
```

Denotamos por `L(B)` (a linguagem de `B`) o conjunto de todas as strings the terminais que podem ser obtidas aplicando-se produções a partir de `B`. Em nosso caso, `L(B)` é a linguagem das seqüências balanceadas de parênteses.

Gramáticas e linguagens de programação

Linguagens de programação como C podem ser definidas através de gramáticas livres de contexto. A definição é, claro, muito mais complexa do que para seqüências balanceadas de parênteses, mas a idéia é a mesma.

Por exemplo, um pedaço da definição da linguagem C poderia ser:

```
Expression = (* alguma coisa *) ;
Statement  = "if" , "(" , Expression , ")" ,
Statement , "else" , Statement ;
Statement  = "while" , "(" , Expression , ")" ,
Statement ;
```

Você pode encontrar aqui a gramática completa da linguagem C, descrita de uma forma um pouco diferente da que usamos acima entretanto.

Um parser para seqüências balanceadas

Com base na gramática acima para seqüências balanceadas de parênteses podemos escrever um programa que testa se uma seqüência de parênteses é balanceada ou não. Um tal programa, que testa se uma string pertence à linguagem definida por uma gramática, é chamado de *parser*.

Existem vários tipos de parsers. O parser que vamos escrever para seqüências balanceadas é um *recursive-descent parser*. Ele consiste de um conjunto de funções mutuamente recursivas, cada uma implementando as produções que começam com um determinado não-terminal. Em nosso caso, temos apenas um não-terminal, `B`. Assim, temos apenas uma função, que chamamos de `B_prod`.

Suponha que queiramos determinar se uma dada seqüência é ou não balanceada. Vamos percorrê-la da esquerda para a direita, guardando numa variável global `cur_char` um ponteiro para o caractere a ser considerado no momento:

```
1 static char *cur_char;
```

Uma chamada à função `B_prod` *consome* da string que começa em `cur_char` o maior prefixo balanceado. Aqui, *consumir* significa que `cur_char` é avançado até o primeiro caractere após o trecho balanceado.

Por exemplo, se `cur_char` aponta para o primeiro caractere de `(()()(`, então após uma chamada à `B_prod`, `cur_char` aponta para o último `(`.

Como podemos implementar `B_prod`? Podemos fazer assim:

- Se `*cur_char == '('`, então aplicamos a primeira produção. Assim, consumimos o caractere `(`, chamamos `B_prod` recursivamente para consumir o maior prefixo balanceado, e verificamos se o próximo caractere é `)`. Se o próximo caractere for diferente de `)`, temos um erro de sintaxe. Se não, consumimos o próximo caractere e chamamos `B_prod` novamente.
- Se `*cur_char != '('`, então aplicamos a segunda produção, sem consumir portanto nenhum caractere.

Abaixo, veja o código da função `B_prod`. A função consome o maior prefixo balanceado a partir de `cur_char`, avançando `cur_char` para o primeiro caractere após o prefixo. Ela devolve não-zero se obtiver sucesso, zero se detectar um erro de sintaxe ao aplicar a primeira produção como explicado acima:

```
1  // Pointer to current character in string being
   parsed.
2  static char *cur_char;
3
4  int B_prod()
5  {
6    if (*cur_char == '(') {
7      cur_char++;
8
9      if (!B_prod()) return 0;
10
```

```
    if (*cur_char != ')')
      return 0;

    cur_char++;

    return B_prod();
  }

  return 1;
}
```

Note que, quando uma chamada recursiva à `B_prod` falha, nós devolvemos zero imediatamente para sinalizar o erro de sintaxe. Assim, `cur_char` não é mais alterado e contém um ponteiro para o caractere no qual o erro foi detectado.

Agora, podemos usar a função `B_prod` para decidir se uma string é balanceada. A função `main` abaixo lê strings da entrada padrão e decide se elas são balanceadas ou não. Caso a string não seja balanceada, mostramos o primeiro caractere onde ocorreu um erro de sintaxe:

```
int main()
{
  char buffer[256];

  while (scanf("%255s", buffer)) {
    cur_char = buffer;

    if (!B_prod() || *cur_char) {
      printf("String is not balanced:\n");
      printf("    %s\n", buffer);

      for (int i = 0; i < cur_char - buffer;
i++)
        putchar(' ');

      printf("    ^\n");
    }
    else
      printf("String is balanced\n");
```

```
19    }
20
21    return 0;
22  }
```

Observe que chamamos `B_prod` para cada string lida. Se `B_prod` devolve zero, ocorreu um erro de sintaxe. Mesmo que `B_prod` devolva um número diferente de zero, a string lida pode não ser balanceada. Por exemplo, ela pode ter um prefixo balanceado mas não ser balanceada, como a string `(()())(`. Assim, é preciso também verificar se, quando `B_prod` devolve não-zero, temos que `*cur_char == '\0'`, ou seja, a string inteira foi consumida. Em ambos os casos de erro, `cur_char` contém um ponteiro para o primeiro caractere no qual o erro foi detectado, e usamos esse ponteiro para imprimir uma indicação de onde ocorreu o erro.

Você pode baixar o programa inteiro `aqui`. Veja alguns exemplos de entrada e saída do programa (entrada em negrito):

```
1  ()(()())()
2  String is balanced
3  ()(()))()
4  String is not balanced:
5      ()(()))()
6            ^
7  ()(()))(
8  String is not balanced:
9      ()(()))(
10             ^
```

Essa estratégia sempre funciona?

Não é para toda gramática que podemos fazer um parser como o acima. A mesma linguagem pode ser representada por gramáticas diferentes, e para algumas delas um *recursive-descent parser* pode não ser possível.

Por exemplo, a linguagem das seqüências balanceadas também pode ser representada pela seguinte gramática:

```
B = B , "(" , B , ")" ;
B = "" ;
```

A primeira produção acima apresenta o fenômeno de ***recursão à esquerda***. Se tentarmos escrever uma função `B_prod` baseando-nos nessa gramática, teremos logo de início uma chamada recursiva à `B_prod`, e o parser vai entrar num laço infinito.

Exercícios

1. Usando o exemplo anterior, crie um programa para aplicar as regras de produção para uma gramática que descreve a linguagem composta de todas as strings da forma `A...AB...B`, compostas de um certo número de letras `A` seguidas do mesmo número de letras `B`, por exemplo `AB`, `AABB`, `AAAABBBB`, etc. Note que a palavra vazia também pertence à linguagem.

```c
#include<stdio.h>

static char *cur_char;
static int b;

int B_prod()
{
  if (*cur_char == 'A' && b==0) {
    cur_char++;
    //B_prod == 1 (!0==1)
    if (!B_prod()) return 0;

    if (*cur_char != 'B')
      return 0;
    b=1;
    cur_char++;

    return B_prod();
  }

  return 1;
}
```

```c

int main()
{
  char buffer[256];

  while (scanf("%255s", buffer)) {
    cur_char = buffer;
    b=0;
    if (!B_prod() || *cur_char) {
      printf("Palavra inválida:\n");
      printf("    %s\n", buffer);

      for (int i = 0; i < cur_char - buffer;
i++)
        putchar(' ');

      printf("    ^\n");
    }
    else
      printf("Palavra válida\n");
  }

  return 0;
}
```