Neste tutorial será criado, com instruções passo a passo, um pequeno compilador, ou para ser mais preciso um interpretador, para expressões numéricas.

Neste interpretador serão aceitas espressões com números, as quatro operações básicas e parênteses. Multiplicação e divisão deverão ter prioridade maior que adição e subtração.

Serão detalhadas as especificações léxica e sintática, será implementado o analisador semântico e por fim será feito um programa para testar os analisadores.

Primeiros Passos

Especificação Léxica

Especificação Sintática

Implementação do Semântico

Criando um programa de teste

Assim que se abre o GALS, ele está pronto para a criação de um analisador léxico e sintático em conjunto. Se você for em Ferramentas -> Opções você podera configurar isso.

Entre as opções disponíveis estão:

- Quais analisadores gerar (léxico, sintático ou ambos)
- Em que linguágem gerar código (Java, C++ ou Delphi)
- Nomes das classes geradas
- Forma de entrada do analizador léxico
- Técnica utilizada pelo analisador sintático

Para este tutorial serão feitos o analisador léxico e o sintático, e os exemplos de código serão em Java, mas em alguns casos serão também dados exemplos em C++ e em Delphi para mostrar as diferênças.

A primeira coisa a se fazer em uma especificação léxica e saber quais os *tokens* que deverão ser reconhecidos pelo analisador.

Como neste exemplo vão ser precisos números, operadores e parênteses, já é possível ter uma idéia de quais tokens serão precisos:

- NUMERO
- +
- -
- *
- /
- (
-)

Antes de especificar de fato so *tokens*, é preciso notar que a especificação léxica é dividida em duas partes: Definições Regulares e Definição dos Tokens.

Os *tokens* são definidos na segunda parte. Nas definições regulares são definidas expressões auxiliáres, para serem utilizadas na definição dos *tokens*.

Definições Regulares

Pare este exemplo será feira apenas a seguinte definição:

```
D : [0-9]
```

Esta definição diz que D (digito) é qualquer letra entre 0 e 9.

Tokens

Os *tokens* para este exemplo são definidos da seguinte forma:

```
"+"
"-"
"*"
"/"
"("
")"
NUMERO : {D}+
 : [\s\t\n\r]*
```

Primeiro são definidos os operadores. Uma grupo de caracteres entre aspas define um *tokens* cuja representação é a de string entre aspas.

Em seguida é definido NUMERO. Para este *token* é fornecida uma expressão regular para representá-lo. Nesta expressão é utilizada a definição regular anteriormente definida. Um NUMERO é um D (digito) repetido uma ou mais vezes. Para utilizar uma definição deve-se colocá-la entre { e }.

Por fim é descrita uma expressão sem um *token* associado. Isto indica ao analisador que ele deve ignorar esta expressão sempre que encontrá-la. Neste caso devem ser ignorados espaço em branco (\s), tabulação (\t) e quebra de linha (\n e \r).

Expressões Regulares

Esta tabela ilustra as possíbilidades de expressões regulares. Quaisquer combinações entre estes padrões é possível. Espaços em branco são ignorados (exceto entre " e ").

a	reconhece a
ab	reconhece a seguido de b
a|b	reconhece a ou b
[abc]	recohece a, b ou c
[^abc]	reconhece qualquer caractere, exceto a, b e c
[a-z]	reconhece a, b, c, ... ou z
a*	reconhece zero ou mais a's
a+	reconhece um ou mais a's
a?	reconhece um a ou nenhum a.
(a|b)*	reconhece qualquer número de a's ou b's
.	reconhece qualquer caractere, exceto quebra de linha
\123	reconhace o caractere ASCII 123 (decimal)

Os operadores posfixos (*, + e ?) tem prioridade máxima. Em seguida está a concatenação e por fim a união (|). Parenteses podem ser utulisador para agrupar símbolos e driblar prioridades.

Caracteres especiais

Os caracteres " \ | * + ? ( ) [ ] { } . ^ - possuem significado especial. Para utilizá-los como caracteres normais deve-se precedê-los por \, ou colocá-los entre " e ". Qualquer sequancia de caractateres entre " e " é tratada como caracteres ordinários.

\+	reconhece +
"+*"	reconhece + seguido de *
"a""b"	reconhece a, seguido de ", seguido de b
\"	reconhece "

Existem ainda os caracteres não imprimíveis, representados por sequancias de escape

\n	Line Feed
\r	Carriage Return
\s	Espaço
\t	Tabulação
\b	Backspace
\e	Esc

\XXX	O caractere ASCII XXX (XXX é um número decimal)

Outras Formas de especificar *tokens*

Pode-se definir ainda um *tokens* como sendo um caso particular de um outro *token*. Por exemplo:

```
ID : [a-z A-Z][a-z A-Z 0-9]* //letra seguida de zero ou mais letras ou dígito

BEGIN = ID : "begin"
END   = ID : "end"
WHILE = ID : "while"
```

Assim define-se que BEGIN, END e WHILE são casos especiais de ID. Sempre que o analisador encontrar um ID ele procura na lista de casos especiais para ver se este ID não é um BEGIN ou um WHILE.

A especificação sintática é feita de produções. As produções para uma gramática para expressões números da forma especificada ficam da seguinte forma:

```
<E> ::= <E> "+" <T>
      | <E> "-" <T>
      | <T>;
<T> ::= <T> "*" <F>
      | <T> "/" <F>
      | <F>;
<F> ::= "(" <E> ")" | NUMERO;
```

Podem ser utilizados na gramática qualquer *token* já declarado como símbolo terminal. Os símbolos não-terminais precisam ser previamente declarados em sua área específica.

Esta gramática possui recursões à esquerda e não está fatorada. Não é possível processá-la com um analisador preditivo sem que antes a gramática seja transformada. Neste exemplo será feito um analisador SLR, portanto a gramática já está pronta.

Inserindo as ações semânticas na gramática ela fica assim:

```
<E> ::= <E> "+" <T> #2
      | <E> "-" <T> #3
      | <T>;
<T> ::= <T> "*" <F> #4
      | <T> "/" <F> #5
      | <F>;
<F> ::= "(" <E> ")" | NUMERO #1;
```

As ações são distribuidas já pensando-se na análise semântica. A ação 1 acontece após um NUMERO ser encontrado. Sua implementação irá calcular o valor numérico do *token* NUMERO e empilhá-lo.

As demais ações irão desempilhar dois valores, efetuar uma operação sobre eles e empilhar o resultado.

É gerada uma classe para o analisador semântico. Sua implementação porém é por conta do usuário.

O único método que o analisador semântico possíu é o método *executeAction()*. O analisador sintático chama ele sempre que uma ação semântica é encontrada. São passados de paâmetro para este método o número da ação semântica que o disparou, e o último *token* reconhecido antes da ação.

Para este exemplo, o analisador semântico vai precisar apenas de uma pilha para avaliar as expressões. Em casos mais complexos, como um compilador, será preciso uma tabela de símbolos também. E o gerador de código deve ser acionado por ações semânticas também.

```
import java.util.Stack;

public class Semantico implements Constants
{
  Stack stack = new Stack();

  public int getResult()
  {
    return ((Integer)stack.peek()).intValue();
  }

  public void executeAction(int action, Token token) throws SemanticError
  {
    Integer a, b;

    switch (action)
    {
      case 1:
        String tmp = currentToken.getLexeme();
        if (tmp.charAt(0) == '0')
          throw new SemanticError("Números começados por 0 não são permitidos",
token.getPosition());
        stack.push(Integer.valueOf(tmp));
        break;
      case 2:
        b = (Integer) stack.pop();
        a = (Integer) stack.pop();
        stack.push(new Integer(a.intValue() + b.intValue()));
        break;
      case 3:
        b = (Integer) stack.pop();
        a = (Integer) stack.pop();
        stack.push(new Integer(a.intValue() - b.intValue()));
        break;
      case 4:
        b = (Integer) stack.pop();
        a = (Integer) stack.pop();
        stack.push(new Integer(a.intValue() * b.intValue()));
        break;
      case 5:
        b = (Integer) stack.pop();
        a = (Integer) stack.pop();
        stack.push(new Integer(a.intValue() / b.intValue()));
        break;
    }
  }
}
```

Este é um exemplo simples, todas as ações são implementadas diretamente dentro do *switch*. Um caso mais

complexo deve ter um método para cada ação, utilizando o *switch* para selecionar o método.

Foi feita uma restrição semântica para exemplificar como devem ser indicados os erros semânticos. Se o analisador encontra um erro, ele deve lançar um *SemanticError*, passando como parâmetro a mensagem de erro e a posição do erro (que é em geral a posição do último *token*).

Java	`throw new SemanticError(msg, pos)`
C++	`throw SemanticError(msg, pos)`
Delphi	`raise ESemanticError.create(msg, pos)`

Este é um programa para testar os analisadores. Ele lê uma expressão da entrada, e imprime seu resultado em seguida.

```
public class Main
{
  public static void main(String[] args)
  {
    Lexico lexico = new Lexico();
    Sintatico sintatico = new Sintatico();
    Semantico semantico = new Semantico();

    LineNumberReader in = new LineNumberReader(new InputStreamReader(System.in));
    String line = in.readLine();

    lexico.setInput( line );

    try
    {
      sintatico.parse(lexico, semantico);
      System.out.println(" = ");
      System.out.println(trans.getResult());
    }
    catch ( LexicalError e )
    {
      e.printStackTrace();
    }
    catch ( SintaticError e )
    {
      e.printStackTrace();
    }
    catch ( SemanticError e )
    {
      e.printStackTrace();
    }
  }
}
```

Para utilizar os analisadores gerados pelo GALS, deve seguir os seguintes passos:

- Instanciar um objeto de cada analisador
- Passar o texto de entrada para o léxico
- Chamar o método *parse* do sintático, tratando os possíveis erros

Em Java

```
Lexico lexico = new Lexico();
Sintatico sintatico = new Sintatico();
Semantico semantico = new Semantico();

...

lexico.setInput( /* entrada */ );

try
{
    sintatico.parse(lexico, semantico);
}
catch ( LexicalError e )
{
```

```
    //Trada erros léxicos
}
catch ( SintaticError e )
{
    //Trada erros sintáticos
}
catch ( SemanticError e )
{
    //Trada erros semânticos
}
```

Em C++

```
Lexico lexico;
Sintatico sintatico;
Semantico semantico;

...

lexico.setInput( /* entrada */ );

try
{
    sintatico.parse(&lexico, &semantico);
}
catch ( LexicalError &e )
{
    //Trada erros léxicos
}
catch ( SintaticError &e )
{
    //Trada erros sintáticos
}
catch ( SemanticError &e )
{
    //Trada erros semânticos
}
```

Em Delphi

```
lexico : TLexico;
sintatico : TSintatico;
semantico : TSemantico;

...

lexico := TLexico.create;
sintatico := TSintatico.create;
semantico := TSemantico.create;

...

lexico.setInput( /* entrada */ );

try
    sintatico.parse(lexico, semantico);
except
  on e : ELexicalError do
    //Trada erros léxicos
```

```
  on e : ESintaticError do
    //Trada erros sintáticos

  on e : ESemanticError do
    //Trada erros semânticos

end;

...

lexico.destroy;
sintatico.destroy;
semantico.destroy;
```

Mesagens de Erro

São geradas mensagens de erro *default* para os possívies erros. Em alguns casos elas podem ser apropriadas, mas em geral você vai querer alterá-las para informar ao usuário uma mensagem mais adequada. As tabelas de erro estão nos arquivos com as constantes.