GALS

Gerador de Analisadores Léxicos e Sintáticos

Introdução

GALS é um ambiente para a geração de analisadores léxicos e sintáticos, desenvolvido por Carlos Eduardo Gesser como trabalho de conclusão de curso do Curso de Bacharelado em Ciências da Computação, da Universidade Federal de Santa Catarina, sendo desenvolvido sob a orientação do Prof. Olinto José Varela Furtado.

Esta ferramenta pode ser obtida em http://gals.sourceforge.net.
GALS é uma ferramenta de Software Livre. Seu código fonte é liberado sob a Licença Publica GNU. (http://www.gnu.org).

Topo

Opções

As opções permitem configurar aspectos dos analisadores gerados.

Opções Gerais

Gerar

- Analisador Léxico
- Analisador Sintático
- Analisadores Léxico e Sintático

Esta opção permite definir o modo de trabalho do gerador. Escolhendo a opção *Analisador Léxico*, será gerado apenas um analisador léxico, sendo pedido que o usuário entre com a definições dos aspéctos léxicos.

A opção *Analisador Sintático* gera um analisador sintático completo, e analisadores léxico e semântico vazios, que devem ser implementados pelo usuário. Neste modo o usuário entra apenas com os aspectos sintáticos.

A última opção faz com que gera gerado um analisador léxico em conjunto com um sintático. Será gerado ainda um analisador semantico vazio, para a implementação do usuário.

Linguagem

- Java
- C++
- Delphi

Está opção permite ao usuário escolher a linguagem em que o analisador será gerado.

Classes

- Analisador Léxico
- Analisador Sintático
- Analisador Semântico
- Package/Namespace

Estas são opções para que o usuário possa ter controle sobre os nomes das classes geradas.
Para Java e C++ pode-se especificar a *package* e o *namespace* respectivamente onde serão geradas as classes (o que é recomendavel).

Topo

Opções do Analisador Léxico

Forma de Entrada

- Stream
- String

Pode-se escolher de onde o analisador léxico gerado irá obter seu fluxo de caracteres: de uma classe de *stream* (geralmente utilizada para se ler diretamente de um arquivo) ou de uma *string* contendo toda a entrada a ser processada (que pode ser obtida de um componente de interface gráfica, por exemplo).

Implementação do Autômata

- Tabela Completa
- Tabela Comprimida
- Específica

O analisador léxico é implementado sobre um autômato finito. Esta opção permite escolher sua forma de implementação. As duas primeiras opções geram um analisador genérico, com uma tabela de transições. A primeira opção gera uma tabela simples, que permite o acesso mais rápido, a custa de espaço (na grande maioria dos casos a tabela gerada é bem esparsa).
A segunda opção gera uma tabela comprimida, mais eficiente em termos de espaço (em casos de tabelas esparsas), mas de desempenho inferior em tempo de busca.
A última opção gera um analisador específico, com as transições programadas diratamente dentro do analisador léxico, o que pode serar um analisador bem eficiente. A desvantagem desta opção é que qualquer alteracão na especificação léxica requer a recompilação do analisador, enquanto que nas outras opções basta recompilar o arquivo de constantes.

Diferencias maiúscula/minúscula em casos especiais

O analisador gerado sempre passa os tokens ao sintático (que passa ao semântico) exatamente como eles estavam no texto original, sem qualquer conversão entre maiúsculas e minúsculas.
Se se pretende gerar um analisador que não faça diferenciação entre maiúsculas minúsculas para os identificadores, esta capacidade deve ser programada em nível semântico.
Então para que serve esta opção?

Esta opção tem a ver com casos especiais, utilizados (geralmente) para a definição de palavras chave. Com esta opçào desabilitada, tanto *Begin* quanto *begin* quanto *BEGIN* seriam reconhecidos como o mesmo token, caso se tenha um caso especial de identificador com a representação igual a *begin*.

Topo

Opções do Analisador Sintático

Classe do Analisador Sintático

- Descendente Recursivo
- LL(1)

- SLR(1)
- LALR(1)
- LR(1) Canônico

Esta opção controla o tipo do analisador seintático que será gerado. Algumas classes impõem restrições sobre a gramática que aceitam, que devem ser observadas quando se for descrevê-la

Topo

Definição dos Aspectos Léxicos

Os aspectos léxicos de uma especificação GALS são definidos pela declaração dos Tokens e pela declaração de Definições Regulares.
Os Tokens definem quais construções léxicas serão aceitas pelo Scanner e o valor léxico de cada uma destas

construções.
As Definições Regulares atuam como expressões auxiliares para a definição de Tokens.

O Analisador Léxico gerado utilizará as expressões regulares dos token para a criação de um Autômato Finito com múltiplos estados finais, cada um deles correspondendo a um dos Tokens declarados.

Topo

Definições Regulares

Uma definição sempre segue a forma:

```
[identificador] : [expressão regular]
```

Cada linha do campo de definições pode conter apenas uma definição regular.
As definições aqui declaradas poderão ser utilizadas em outras expressões regulares, utilizando seu identificador entre { e }.

Topo

Tokens

Antes de se mostrar como são declarados os Tokens é preciso se dar uma pequena explicação sobre o funcionamento do analisador léxico.

O analisador gerado funciona simulando um autômato finito, rodando em cima de uma tabela de transições. O analisador verifica o próximo caractere da entrada e o estado atual do autômato (inicialmente zero) e move o autômato para seu próximo estado.
Se eventualmente o autômato chegar a um estado final, sempre correspondente a um token, ele ainda não pode dar a análise deste token como concluída, pois é preciso tentar identificar a seqüência de caracteres mais longa possível (um analisador para Pascal não pode identificar o token ":" no momento que encontrar este caractere, ele precisa continuar, pois pode ser que o token seja ":=").
Assim, o analisador somente para quando não consegue mais prosseguir na tabela de transições. Se durante este processo ele encontrou algum token, ele produz o token correspondente ao último estado final alcançado (a seqüência mais longa de caracteres). Se nenhum token foi encontrado então um erro léxico é gerado.

Existem diversos meio para a definição de Tokens, cada um deles mais adequado a um tipo de Token.
Como nas Definições Regulares, aqui também só é permitida a declaração de um token por linha.

Definindo Token a partir do identificador

Pode-se declarar um token apenas declarando um identificador para ele. Esta é a forma mais simples de se declarar um token, porém a menos flexível.
Pode-se utilizar duas formas de identificadores:

- Identificadores Normais
- Qualquer sequência de caracteres entre aspas (")

Um token declarado desta forma irá gerar um autômato finito que identifica o próprio identificador, ou seja, sempre que o analisador encontrar a sequência de caracteres relativa ao identificador ele produzirá o Token correspondente. Por exemplo:

begin identificaria begin, e
"!=" identificaria !=.

Definindo Token a partir de uma Expressão Regular

Esta é a forma mais genérica de se definir um token. Seu caso mais geral é idêntico à declaração de uma Definição Regular:

```
[identificador] : [expressão regular]
```

Sempre que o analisador identificar a expressão regular ele produzirá o token correspondente.

Pode-se especificar para um token uma segunda expressão regular, chamada neste caso de contexto.

```
[identificador] : [expressão regular] ^ [expressão de contexto]
```

Se um contexto for especificado, sempre que o token vier a ser identificado, o analisador tenta analisar a expressão de contexto. Se a expressão não puder ser encontrada após o token o analisador considera este token como inválido (como se chegasse a um ponto sem transição possível na tabela de transições).
Esta construção pode ser entendida como: somente identifique este token se, depois dele, for possível identificar a expressão de contexto.
O contexto é analisado, porém não é consumido pelo analisador léxico.

Podem existir casos em que ao ser encontrado um erro (nenhuma transição possível), este deve ser reportado de qualquer forma, mesmo que durante a análise deste token tenha-se encontrado outros tokens validos possíveis.
Por exemplo: em uma linguagem com comentários delimitados por "(*" e "*)", um comentário não fechado seria um erro. Este erro fará com que o analisador verifique se durante a análise ele não encontrou nenhum outro token valido. Se a linguagem também possuir a declaração de um token correspondente a "(", o analisador o teria encontrado nesse processo, o retornaria, continuando a análise a partir do "*" do comentário. Para prevenir isto, o token correspondente ao comentário deveria ser declarado desta forma:

```
[identificador] :! [expressão regular]
```

Um token declarado deste modo não verifica outros tokens validos encontrados antes em caso de erro.

Nos três casos, o identificador pode ser omitido (a declaração começa diretamente pelo ":" ou ":!").
Quando o identificador não é fornecido, o analisador gerado passa a ignorar tokens gerados pela expressão regular correspondente.
Deste modo pode-se fornecer expressões para comentários e caracteres de espaço em branco (espaço, quebra de linha, tabulação, etc.) para que o analisador gerado ignore.

Definindo Token como caso especial de outro Token

Pode-se definir um token como sendo um caso particular de um outro token base.
Nestes casos, sempre que o analisador identifica o token base, ele procura pelo valor do token em uma lista de casos especiais. Se for encontrado, o caso especial é retornado, senão é produzido o token base. Esta declaração é feita da seguinte forma:

```
[identificador] = [token base] : [valor]
```

onde token base é um token declarado previamente, e valor é uma sequência de caracteres entre aspas.

Como pode-se deduzir, esta construção é especialmente útil para a declaração das palavras reservadas de uma linguagem. Em geral, as palavras reservadas seguem o mesmo padrão dos identificadores.
Utilizar esta construção faz com que o autômato gerado seja bem menor de que se cada caso especial fosse declarado como um token comum. A lista dos casos é gerada em ordem, e a localização de um caso é feita por busca binária.

Topo

Expressões Regulares

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

a	reconhece a
ab	reconhece a seguido de b
a|b	reconhece a ou b
[abc]	reconhece 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	reconhece o caractere ASCII 123 (decimal)

Os operadores posfixos (*, + e ?) tem a maior prioridade. Em seguida está a concatenação e por fim a união (|). Parênteses podem ser utilizador para agrupar símbolos.

Os caracteres `" \ | * + ? ( ) [ ] { } . ^` - possuem significado especial. Para utilizá-los como caracteres normais deve-se precedê-los por \, ou colocá-los entre aspas. Qualquer seqüência de caracteres entre aspas 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 seqüências 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)

Topo

Exemplo

Definições Regulares

```
L  : [A-Za-z]
D  : [0-9]
WS : [\ \t\n\r]
COMMENT : "(*" [^ "*" ]* "*)"
```

Tokens

```
//pontuação
"("
")"
";"

//tokens
id : {L} ( {L} | {D} | _ )*
num : {D}+ ^ [^ {L} ]//um ou mais dígitos, seguido de qqr char menos letra

//palavras chave
begin = id : "begin"
end   = id : "end"
if    = id : "if"
then  = id : "then"
else  = id : "else"
while = id : "while"
do    = id : "do"
write = id : "write"

//ignorar espaços em branco e comentários
 : {WS}*
 :! {COMMENT}
```

Topo

Definição dos Aspectos Sintáticos

Os Aspectos Sintáticos são compostos pela declaração de Símbolos Terminais (Tokens), Símbolos Não-Terminais e Produções.

Símbolos Terminais (Tokens)

O modo como estes símbolos são declarados depende do tipo de projeto que esta sendo feito.

Analisadores Léxico e Sintático conjuntos

Neste tipo de projeto os Tokens declarados na Especificação Léxica são tomados como Símbolos Terminais.

Analisador Sintático sozinho

Aqui os Tokens devem ser declarados explicitamente. Um Token pode ser declarado desta forma pode ser definido por um Identificador, ou uma expressão qualquer entre aspas. Cada linha deve conter a declaração de apenas um Token.

Topo

Símbolos Não-Terminais

O Símbolos Não-Terminais devem ser todos declarados antes de poderem ser utilizados em produções. Sua forma é a de um identificador entre < e >. Assim como no caso dos Tokens, apenas um símbolo pode ser declarado por linha.

O primeiro símbolo declarado é considerado o símbolo inicial da gramática.

Topo

Produções

A declaração das produções segue um formato baseado na notação BNF:

```
<não-terminal> ::= <lista de símbolos> | <lista de símbolos> | ... ;
```

Pode-se agrupar as definições ou deixá-las separadas, ou seja:

```
<A> ::= <B> | <C> ;
```

é igual a:

```
<A> ::= <B> ;
<A> ::= <C>;
```

```
Somente são aceitos nas produções símbolos já previamente declarados. O uso de um símbolo
(terminal ou não=terminal) não declarado gera um erro semântico.
Uma exceção a essa regra diz respeito ao símbolo terminal especial î (letra i, com um acento
circunflexo), que representa o símbolo epsilon (sentença vazia).
```

Topo

Restrições

Existem restrições impostas à forma das produções, de acordo com a classe de analisador sintático que se

pretende gerar. Para Analisadores Descendentes (LL e Descendente Recursivo) não é permitido que a gramática possua Recursão à Esquerda ou que não esteja na sua Forma Fatorada. A tentativa de se gerar um analisador com uma gramática neste estado resultará em erro. A terceira restrição para gramáticas LL é checada, mas não impede que seja gerado o analisador. Enquanto as duas outras restrições podem ser facilmente removidas aplicando-se algoritmos de transformação à gramática, esta ultima não o é. Gramáticas com este problema são ambíguas, e durante a geração do analisador será pedido ao usuário para indicar qual produção deve ser escolhida para eliminar a ambiguidade.

Analisadores Ascendentes (LR, LALR e SLR) não possuem problemas com recursão à esquerda ou fatoração, mas mesmo assim não conseguem tratar gramáticas ambiguas. Neste caso, assim como nas analisadores descendentes, o usuário deverá escolher em casos de ambiguidade entre empilhar um símbolo ou reduzir por uma produção, ou então entre duas produções atraves das quais se pode reduzir.

Topo

Exemplo

Tokens

```
"("
")"
";"
id
num
begin
end
if
then
else
while
do
write
```

Não-Terminais

```
<C>
<C_LIST>
<IF>
<ELSE>
<WHILE>
<WRITE>
<E>
```

Produções

```
<C> ::= <IF>
      | <WHILE>
      | <WRITE>
      | begin <C_LIST> end;
<C_LIST> ::= <C> ";" <C_LIST>
      |î;
<IF> ::= if <E> then <C> <ELSE>;
<ELSE> ::= else <C>
      | î;
<WHILE> ::= while <E> do <C>;
```

```
<WRITE> ::= write "(" <E> ")";
<E> ::= id
      | num;
```

Obs. 1: Pode-se utilizar neste exemplo os tokens do exemplo do analisador léxico.
Obs. 2: Esta gramática é ambígua. Para gerar-se um analisador descendente é preciso escolher a produção `<ELSE> ::= else <C>` durante a etapa de resolução de ambigüidades, e para analisadores ascendentes deve-se escolher empilhar "else".

Topo

Definição dos Aspectos Semânticos e de Geração de Código

Aspectos Semânticos são definidos atraves da introdução de Ações Semânticas dentro das produções da especificação sintática. Estas ações são da forma:

```
#<número>
```

Durante a analise sintática, ação semânticas instruem o analisador sintático a envocar o analisador semântico, passando-lhe como parâmetros o número da ação, e o mais recente token produzido pelo analisador léxico. Cabe ao usuário implementar as ações semânticas, na linguagem de destino.

Estas ações podem ser responsáveis por tarefas de análise semântica (adicionar algum símbolo à tabela de símbolos, checar tipos, verificar se um símbolo ja foi declarado, etc) ou pela geração de código (fazendo-se com que a ação semântica chame o gerador de código).

Topo

Exemplo

Foram colocadas na gramática do exemplo anterior algumas ações. Cabe ao usuário dar sentido a elas, implementando-as.

A ação 2 poderia ser resposável por checar o tipo da expressão e gerar o código para imprimir seu valor.

```
<C> ::= <IF>
      | <WHILE>
      | <WRITE>
      | begin <C_LIST> end;
<C_LIST> ::= <C> ";" <C_LIST>
      | î;
<IF> ::= if <E> #1 then <C> <ELSE>;
<ELSE> ::= else <C>
      | î;
<WHILE> ::= while <E> #1 do <C>;
<WRITE> ::= write "(" <E> #2 ")";
<E> ::= id #3
      | num #4;
```

Topo

Utilização do Código Gerado

Será demonstrada agora a forma de utilização dos analisadores.

Os exemplos serão dados em Java, o uso para as outras linguagens é análogo (exceto em casos especiais, que serão demonstrados).

Topo

Analisador Léxico

Um analisador léxico possui os seguintes métodos:

`Lexico();`
: Construtor padrão

`Lexico(String input);`
: Construtor de inicialização

`void setInput(String input);`
: Método para passar a entrada ao analisador

`void setPosition(int pos);`
: Método para indicar a posição a partir da qual o próximo token deve ser procurado

`Token nextToken() throws LexicalError;`
: Método chamado para se obter o próximo token da entrada

No caso de estar utilizando *streams* como forma de entrada, o construtor de inicialização e o método *setInput* receberão como parâmetro um objeto da classe *Reader* (*istream* em c++ e *TStream* em Delphi) em vez de uma string.

O método *nextToken* é o pricipal desta classe. A cada chamada o analisador tenta identificar um token a partir da posição atual na entrada. Existem três resultados possíveis:

Um token é encontrado
: Neste caso, é retornado um novo objeto da classe Token.

A posição de leitura era o fim da entrada
: Neste caso é retornado *null* ao chamador, indicando o fim do fluxo de tokens

Nenhum token reconhecido
: Se nenhum token foi reconhecido pelo analisador, será lançada uma exceção

A cada chamada com sucesso, um novo token é alocado. Em C++ e Delphi ele deve ser desalocado quando não for mais necessário.

O token retornado possui três atributos: seu valor numérico (id), seu valor textual (lexeme) e a posição na entrada onde foi encontrado (position).

Exemplo de uso do Analisador Léxico

Em Java

```
Lexico lexico = new Lexico();
//...
lexico.setInput( /* entrada */ );
//...
try
{
    Token t = null;
```

```
        while ( (t = lexico.nextToken()) != null )
        {
            System.out.println(t.getLexeme());
        }
    }
    catch ( LexicalError e )
    {
        System.err.println(e.getMessage() + "e;, em "e; + e.getPosition());
    }
```

Em C++

```
    Lexico lexico;
    //...
    lexico.setInput( /* entrada */ );
    //...
    try
    {
        Token *t = 0;
        while ( (t = lexico->nextToken()) != 0 )
        {
            std::cout &l;t< t->getLexeme() << '\n';
            delete t;
        }
    }
    catch ( LexicalError &e )
    {
        std::cerr << e.getMessage() << "e;, em "e; << e.getPosition() << '\n';
    }
```

Em Delphi

```
    lexico : TLexico;
    t : TToken;
    //...
    lexico := TLexico.create;
    //...
    lexico.setInput( (* entrada *) );
    //...
    try
        t := lexico.nextToken;
        while (t <> nil)
        begin
            writeln(e.getLexeme);
            t.Destroy;
            t := lexico.nextToken;
        end;
    except
      on e : ELexicalError do
        writeln(e.getMessage, ', em ', e.getPosition);
    end;
    //...
    lexico.destroy;
```

Topo

Analisador Sintático

O analisador sintático possui apenas um método público (além de seu construtor padrão):

```
void parse(Lexico scanner, Semantico semanticAnalyser) throws LexicalError, SyntaticError,
SemanticError
```

Para este método é passado um analisador léxico e um semântico (em c++ via ponteiros).
Se nenhum erro for detectado, o método termina de forma normal (o analisador semântico deve ser programado de forma que ele guarde os resultados finais da análise, se houverem).

Este método é o "coração" do processo de análise, e os erros detectados durante esta devem ser tratados pelo chamador deste método.
Erros léxicos vem do analisador léxico na forma da exceção LexicalError. Erros semânticos serão reportados via a exceção SemanticError. O próprio analisador sintático detecta erros, e lança a exceção SyntaticError quando os encontra.

Interface com o Analisador Léxico

Sempre que um novo token for preciso, o método nextToken do analisador léxico é envocado. É esperado que este método retorne null quando não houverem mais tokens para serem processados, e que lance uma exceção LexicalError quando encontre um erro léxico.
Em C++ e em Delphi, é esperado quea cada chamada o token retornado seja alocado dinamicamente, pois o mesmo será desalocado posteriormente (via delete/Destroy);

Interface com o Analisador Semântico

Sempre que uma acao semântica for necessária, o analisador semântico será chamado, pelo método executeAction, que recebe de parâmetro o número da ação atual, e o mais recente token produzido pelo léxico.
É esperado que o analisador semântico lance um SemanticError quando encontrar uma situação de erro semântico.

Topo

Analisador Semântico

O Analisador Semântico é sempre implementado pelo usuário. Sua interface consiste do método:

```
void executeAction(int action, Token token) throws SemanticError;
```

Os parâmetros indicam a ação semântica que deve ser executada e o mais recente token produzido pelo analisador léxico (em c++ ele é passado via ponteiro).

Pode-se implementar de varios modos este método. Para gramáticas com poucas açãoes semânticas, pode-se construir um switch/case em função do parametro action e colocar o código da ação diretamente dentro deste comando, ou delegar um outro método para executá-la (com certeza mais recomendado)
Em gramáticas com muitas ações, pode ser mais interessante criar um array de *callbacks* , indexado pelo número da ação semântica.

Se um erro semântico for detectado, ele deve ser informado ao analisador sintático lançando uma excessão do tipo SemanticError. Isto é importante para manter a uniformidade na detecção de erros

Topo

Tabelas de Erros

As exceções geradas pelos analisadores léxico e sintático utilizam como mensagens de erro constates literais declaradas nos arquivos ScannerConstants.java e ParserConstants.java, Constants.cpp ou ainda UConstants.pas, dependendo da linguagem objeto.

São geradas mensagens padrão, mas na maioria dos casos mensagens personalisadas irão identificar melhor os erros para o usuário final da aplicação.

Topo

Tratamento de Exceções

Muitas pessoas não estão familiarizadas com o mecanismo de tratamento de excessões utilizado pelo GALS para o tratamento de erros, por isso aqui segue uma breve descrição sobre seu funcionamento.

Lançando uma exceção

Em uma situação de erro pode-se lançar uma exceção, indicando este erro. Isto é feito da seguite forma:

Em Java

```
throw new ClasseDeExcecao(parametros);
```

Em C++

```
throw ClasseDeExcecao(parametros);
```

Em Delphi

```
raise ClasseDeExcecao.Create(parametros);
```

Tratando uma exceção

Quando se executa um pedaço de código que pode eventualmente gerar uma exceção, deve-se tratar esta possível condição da seguinte forma:

Em Java

```
try
{
        //codigo que pode gerar exceção
}
catch (ClasseDeExcessao e)
{
        //trata exceção do tipo ClasseDeExcessao
}
```

Em C++

```
try
{
        //codigo que pode gerar exceção
}
catch (ClasseDeExcessao &e)
{
        //trata exceção do tipo ClasseDeExcessao
}
```

Em Delphi

```
try
        //codigo que pode gerar exceção
except on e : ClasseDeExcessao do
        //trata exceção do tipo ClasseDeExcessao
end;
```

As exceções geralmente possuem atributos que indicam o motivo do erro.

Exceções no GALS

As exceções utilizados no GALS possuem dois atributos: uma mensagem de erro e a posição (na entrada) onde o erro aconteceu. Existem três classes concretas de exceções:

- LexicalError
- SyntaticError
- SemanticError

que são produzidas pelos analisadores léxico, sintático e semântico respectivamente. Existe ainda uma quarta classe: AnalysisError, que serve de base para as outras três. Quando se for tratar os erros gerados pelo método parse do analisador sintático, pode-se tratar cada exceção separadamente, ou tratar todas de uma vez só tratando-se AnalysisError.

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 ( SyntaticError 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 ( SyntaticError &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 : ESyntaticError do
        //Trada erros sintáticos
      on e : ESemanticError do
        //Trada erros semânticos
    end;
    //...
    lexico.destroy;
    sintatico.destroy;
    semantico.destroy;
```

Topo
