UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E APLICADAS
DEPARTAMENTO DE COMPUTAÇÃO E SISTEMAS

DEISE KELLEY SILVA

UM GERADOR DE ANALISADOR SINTÁTICO PARA ADAPTABLE PARSING EXPRESSION GRAMMARS

João Monlevade

2018

DEISE KELLEY SILVA

UM GERADOR DE ANALISADOR SINTÁTICO PARA ADAPTABLE PARSING EXPRESSION GRAMMARS

Monografia apresentada ao curso de Engenharia de Computação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Orientador: Elton Maximo Cardoso

Coorientador: Leonardo Vieira dos Santos Reis

João Monlevade

2018

S586g Silva, Deise Kelley
Um gerador de analisador sintático para adaptable parsing expression grammars. [Manuscrito]./ Deise Kelley Silva. - 2018.
42 f. : il.

Orientador: Prof. Elton Maximo Cardoso.
Coorientador: Prof. Leonardo Vieira dos Santos Reis.
Monografia (curso de graduação em Engenharia de Computação) Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Departamento de Computação e Sistemas de Informação.

1. Computação - Sintaxe 2. Linguagem de programação (Computadores) - Sintaxe. I. Universidade Federal de Ouro Preto. II. Titulo.

CDU 004.43

Catalogação: bibjmv.sisbin@ufop.edu.br

Curso de Engenharia de Computação

FOLHA DE APROVAÇÃO DA BANCA EXAMINADORA

Um gerador de Analisador Sintático para Adaptable Parsing Expression Grammar

Deise Kelley Silva

Monografia apresentada ao Instituto de Ciências Exatas e Aplicadas da Universidade Federal de Ouro Preto como requisito parcial da disciplina CSI496 – Trabalho de Conclusão de Curso II do curso de Bacharelado em Engenharia de Computação e aprovada pela Banca Examinadora abaixo assinada:

Elton M. Cardoso

Prof. Msc. Elton Máximo Cardoso
DECSI - UFOP
Professor Orientador

Prof. Dr. Leonardo Vieira dos Santos Reis
DCC - UFJF
Professor Coorientador

Alexandre Magno de Sou

Prof. Msc. Alexandre Magno de Souza
DECSI - UFOP
Professor Convidado

Prof. Msc. Bruno Cerqueira Hott
DECSI - UFOP
Professor Convidado

Este trabalho é dedicado à Sônia e Silvio,
os principais responsáveis pela concretização desse sonho.

Agradecimentos

Agradeço primeiramente a Deus por me dar forças e sabedoria, sem ele eu não conseguiria ter chegado até aqui. Obrigada Senhor!

Aos meus pais que me apoiaram em toda a minha graduação permitindo que esse sonho se realizasse. Amo eternamente vocês.

Aos meus orientadores Elton Máximo Cardoso e Leonardo Vieira dos Santos Reis, pela oportunidade, dedicação e paciência, além de todo o conhecimento repassado a mim.

A Dasayeve Xavier por todo apoio e carinho. Muito obrigada pela paciência e companheirismo.

Enfim, obrigada a todos que de alguma forma tornaram possível a realização desse trabalho.

"Tudo posso naquele que me fortalece"
(Filipenses 4:13)

Resumo

Adaptable Parsing Expression Grammars (APEG) é um modelo adaptável para especificação de linguagens baseado em Parsing Expression Grammars (PEG) que permite a modificação do conjunto de regras da gramática em tempo de análise do programa de entrada, permitindo especificar os mecanismos de extensibilidade de linguagens extensíveis. Para tal, APEG estende PEG com a noção de atributos. Nesse trabalho foi criado um gerador de analisador sintático para APEG sem a noção de adaptabilidade, ou seja PEG com atributos, no qual recebe uma gramática APEG como entrada e gera automaticamente o analisador sintático escrito na linguagem Java.

Palavras–chave: extensibilidade; sintaxe; PEG; APEG.

Abstract

Adaptive Parsing Expression Grammars (APEG) is an adaptive model for language specification based on Parsing Expression Grammars (PEG) that allows the modification of the set of rules of the grammar in time of analysis of the input program, allowing to specify the mechanisms of extensibility of languages extensible. To this end, APEG extends PEG with the notion of attributes.In this work we have created a syntax analyzer generator for APEG without the notion of adaptability, that is, PEG with attributes, in which it receives an APEG grammar as input and automatically generates the parser written in the Java language.

Key-words: extensiblility; syntax; PEG; APEG.

Lista de ilustrações

Lista de abreviaturas e siglas

PEG	Parsing Expression Grammar
APEG	Adaptable Parsing Expression Grammar
AST	Abstract Syntax Tree (Árvore sintática de abstração)
DSL	Domain-Specific Language (Linguagens de Dominio específico)
CFG	Context Free Grammar (Linguagem Livre de Contexto)

Sumário

1 Introdução

Geradores automáticos de analisadores sintáticos têm sido usados por mais de 50 anos. Ferramentas como o YACC (JOHNSON, 1979) geram, automaticamente, um analisador sintático a partir de uma definição formal da sintaxe da linguagem, que usualmente é baseada em uma gramática livre do contexto. A principal motivação para geradores automáticos de analisadores sintáticos é a construção de um compilador correto e que reconhece todas as sentenças da linguagem que se pretende especificar, visto que com uma implementação manual é difícil garantir que todos os programas de uma linguagem serão corretamente analisados. Apesar das vantagens descritas acima, analisadores sintáticos para linguagens extensíveis (WILSON, 2004) têm sido implementados manualmente (REIS; IORIO; BIGONHA, 2014b).

Linguagens extensíveis (WILSON, 2004) impõem novos desafios para a geração automática de analisadores sintáticos, visto que terão que lidar com adição de novas regras durante a análise do programa de entrada. Tais extensões de regras exigem que um mecanismo de atualização do código do analisador sintáticos dinamicamente. Para realizar a geração automática de analisadores sintáticos de linguagens é necessário um modelo formal de especificação. Em particular, para descrever linguagens extensíveis, o modelo de especificação deve ser capaz de descrever os mecanismos de extensibilidade dessas linguagens. *Adaptable Parsing Expression Grammar (APEG)* (REIS et al., 2012; REIS; IORIO; BIGONHA, 2014a) é um modelo formal para especificação de linguagens baseado em *Parsing Expression Grammars (PEGs)*(FORD, 2004). APEG estende PEG com a noção de adaptabilidade, permitindo estender dinamicamente a sintaxe concreta da linguagem durante a análise do programa de entrada. Usando APEG, linguagens extensíveis podem ser formalmente especificadas, apresentando diversas vantagens (REIS; IORIO; BIGONHA, 2014b).

Os mecanismo de extensibilidade dinâmica de APEG, no entanto, impõem novos desafios na geração automática de analisadores sintáticos eficientes, visto que é necessário lidar com extensões da gramática, necessitando que o código do analisador seja atualizado dinamicamente. Em (REIS et al., 2014), uma estratégia para gerar código a partir de uma especificação APEG, que combina compilação e interpretação, é discutida. REIS et al. propõem gerar código a partir da especificação da gramática e interpretar as regras que são adicionadas dinamicamente.

Nesse trabalho é tratado o problema da geração automática de analisadores sintáticos baseado em APEG. Abordamos a implementação do gerador de PEG com atributos, base para APEG. As questões referentes às modificações dinâmicas das regras não foram

implementadas.

Este documento está organizado da seguinte maneira: o Capítulo 2 apresenta os modelos PEG e APEG. O Capítulo 3 discute as abordagens para geração automática de analisadores sintáticos baseado em PEG. No Capítulo 4 é descrito a implementação do gerador automático de analisador sintático para PEG com atributos e no Capítulo 5 são apresentadas as conclusões

2 Gramáticas Adaptáveis

O termo *linguagem extensível* é usado para se referir à linguagens com construções que permitem estender a própria sintaxe concreta. Essa ideia de oferecer recursos para adicionar novas construções às linguagens não é nova e remota a linguagem Lisp e seus dialetos. Recentemente, o interesse nessa filosofia em linguagens que não adotam o formato *S-expression* tem crescido. Uma das principais motivação é o uso de linguagens de domínio específico como bibliotecas (ERDWEG et al., 2011).

Por exemplo, a Figura 1 apresenta a sintaxe de uma linguagem extensível simples, denominada μSugar. Um programa nessa linguagem começa com uma sequência de declarações de sintaxes, seguido por uma lista de comandos, possivelmente estendidos. Um comando regular, representado por *stmt*, pode ser uma atribuição, um comando de leitura ou escrita de um valor da entrada/saída padrão, um comando condicional ou um comando de repetição. Um bloco de comandos estendido (*extBlock*) começa com uma lista de um ou mais nomes de sintaxes seguidos por um bloco.

A Figura 2 mostra um programa escrito na linguagem μSugar. Nas linhas 1 a 4, duas novas regras são declaradas e recebem o nome de *sfor*. A segunda regra, linha 3, define a sintaxe de um comando *for* e a primeira, line 2, especifica que essa nova sintaxe pode ser usada como um comando da linguagem μSugar. Essa parte do código apenas declara um novo conjunto de regras, porém não modifica efetivamente a gramática original da linguagem.

O código das linhas 6 a 9 é um bloco de comandos estendidos cuja semântica é ativar, durante o processo de análise sintática da entrada, as regras especificas da extensão, nesse exemplo as regras referente à sintaxe *sfor*, efetivamente estendendo a linguagem.

Esse processo também determina o escopo da nova sintaxe. Após a linha 9, o comando *for* não é mais visível. Esse mecanismo de escopo permite definir várias sintaxes e usá-las, combinando-as ou não, em diferentes partes do programa. Embora μSugar seja uma simples linguagem de exemplo, ela apresenta todas as características de extensibilidade presentes na linguagem extensível SugarJ (ERDWEG et al., 2011).

Considerando a teoria de linguagens livre de contexto, a gramática da Figura 1 não gera o programa da Figura 2, pois a nova sintaxe do comando *for* não estava previamente definida na gramática original. De fato, o modelo de gramáticas livres do contexto não oferecem suporte para lidar com modificações no conjunto de regras da gramática durante o processamento da palavra de entrada. Assim, os analisadores sintáticos gerados a partir de CFGs não são capazes de lidar com extensões durante a análise da palavra de entrada.

```
1<Prog>    ::=  <newSyn>* <extStmt>+
2<newSyn> ::= 'define' <sName> '{' <rule>+ '}
3<rule> ::= <ntName> '->' <pattern> ';'
4
5 <pattern> ::= <pattern> <pattern>
6   | <pattern> '/' <pattern>
7   | '!' <pattern>
8   | <pattern> '*'
9   | '(' <pattern> ')'
10  | <ntName>
11  | 'lambda'
12  | LITERAL
13
14 <extStmt> ::= <extBlock> | <stmt>
15 <extBlock> ::= 'syntax' <sName> (',' <sName>)* <block>
16 <block> ::= '{' <stmt>+ '}'
17
18<stmt> ::= <assign> ';'
19   |'print' '(' <expr> ')' ';'
20   | 'read' '(' <var> ')' ';'
21   | 'if' '(' <expr> ')' <block>
22   | 'loop' '(' <expr> ')' <block
23
24<assign> ::= <var> ':=' <expr>
25
26<expr> ::= <expr> '+' <expr>
27   | <expr> '-' <expr>
28   | <expr> '<' <expr>
29   | <expr> '=' <expr>
30   | '(' <expr> ')'
31   | <var>
32   | 'true'
33   | 'false'
34
35<var> ::= ID
36<sName> ::= ID
37<ntName> ::= ID
```

Figura 1 – sintaxe EBNF da linguagem extensível μSugar.

Adaptable Parsing Expression Grammar (APEG) (REIS et al., 2012; REIS; IORIO; BIGONHA, 2014a) é um modelo formal de descrição de linguagem capaz expressar modificações dinâmica no conjunto de regras da gramática. A seguir, o modelo PEG 2.1 é apresentado e, em seguida, na Seção 2.2 discute o modelo APEG.

2.1 PEG

Parsing Expression Grammars (PEG) é um formalismo para descrever reconhecedores de linguagens (FORD, 2004). Diferentemente de CFG, PEG impõe uma ordem determinística de escolha para as alternativas, evitando, assim, ambiguidade. PEG descreve analisadores sintáticos descendentes recursivos com retrocesso limitado definindo como

```
1 define sfor {
2   stmt -> nfor;
3   nfor -> 'for' '(' attr ';' expr ';' attr ')' '{' stmt+ '}';
4 }
5
6 syntax sfor {
7   for (i := 1; i < 10; i := i + 1) {
8     print i;
9   }
10 }
11
12 i := 1;
13 loop (i < 10) {
14   print i;
15   i := i + 1;
16 }
```

Figura 2 – Um exemplo de programa na linguagem μSugar

analisar uma linguagem em vez de como gerar uma linguagem. Por exemplo, sejam as seguintes regras:

$$\mathbf{X <- x \mid xy}$$
$$\mathbf{X <- xy \mid x}$$

Em uma gramática livre de contexto essas regras são equivalentes, ou seja, não há relevância na ordem na qual as regras da variável X são apresentadas. Portanto, a palavra **xy** é gerada por qualquer uma das formas para descrever as regras da variável X. No entanto, em uma gramática PEG essas regras não são equivalentes. A ordem na qual as alternativas são dispostas é relevante. PEG impõe que a ordem na qual as alternativas são "testadas" é a ordem na qual são escritas e uma alternativa só é tentada se as anteriores falharem. Assim, se a palavra de entrada começar com **x**, a primeira forma que as regras de **X** sempre terá sucesso na primeira alternativa e a segunda alternativa nunca é testada.

Sejam e expressões e λ a cadeia vazia. As expressões de parsing em PEG são definida recursivamente como:

- ε : representa uma string vazia
- a : representa um terminal, pode ser qualquer letra que esteja incluida no alfabeto da linguagem
- A : representa todos os não terminais e geradores da linguagem.
- $e_1...e_n$: chama $e_1...e_n$ nessa ordem e retrocede se algum deles falha
- e_1/e_2 : uma escolha priorizada entre alternativas, procura uma alternativa até uma casar

- $e*$: Fecho de Klenee
- $!e$: retorna sucesso se e falhou

A Figura 3 mostra uma gramática em PEG que reconhece a linguagem $L = \{a^n b^n c^n \mid n \geq 1\}$:

```
S <- &(A 'c') 'a'+ B !.
A <- 'a' A? 'b'
B <- 'b' B? 'c'
```

Figura 3 – Peg para $L = \{a^n b^n c^n \mid n \geq 1\}$

Essa linguagem não é uma linguagem livre de contexto pois não pode ser gerado por uma gramática livre de contexto, mas é reconhecida por PEG. Esse exemplo ilustra que há linguagens que não são livres de contexto, mas podem ser reconhecidas por uma gramática PEG. A gramática da Figura 4 descreve a linguagem L= $\{x^n \mid$ n é ímpar $\}$ escrita na sintaxe GLC:

```
S <- 'x' S 'x' | 'x'
```

Figura 4 – GLC para L= $\{x^n \mid$ n é ímpar $\}$

A palavra *xxx* é gerada por essa gramática. No entanto, se interpretamos essas regras como uma PEG, essa mesma palavra não é reconhecida, pois tenta-se sempre a primeira possibilidade, o qual tem sucesso com o primeiro x. Em seguida, entra-se, recursivamente, na regra S que, de modo análogo, reconhece o segundo x e, na sequência, procede como anteriormente. Após três chamadas recursivas da variável S, haverá falha, pois não há mais símbolos da entrada a ser consumido, retrocede-se e tenta a próxima alternativa, que também falhará. Esse processo será repetido suscetivamente ocasionando a falha em reconhecer a palavra de entrada *xxx*. Esse exemplo ilustra que uma gramática livre de contexto não poder ser transformada diretamente em uma gramática PEG.

2.2 APEG

APEG é um formalismo que estende o modelo de PEG (FORD, 2004) com a noção de atributos sintáticos, o que permite a extensão do conjunto de regras da gramática em tempo de análise do programa de entrada. (REIS; IORIO; BIGONHA, 2014a). Assim, a geração de código de um analisador sintático baseado em APEG deve ser capaz de lidar com modificações do conjunto de regras da gramática dinamicamente, o que implica

em possíveis alterações no código gerado, pois códigos para as novas regras devem ser "inseridos" durante a análise de sintática da entrada.

Os *atributos sintáticos* de APEG são valores passados para os não-terminais e usados durante a análise sintática. A própria gramática em APEG são valores de primeira classe passadas para todo não-terminal, as regras usadas durante a análise sintática vêm dessa gramática.A habilidade de passar novos valores de gramáticas é o que permite estender dinamicamente a gramática em APEG.

APEG possui dois tipos de atributos: herdados e sintetizados, que podem ser representado com setas direcionais. Os atributos herdados funcionam como parâmetros das funções e são representados com setas para baixo, já os sintetizados são como retornos e são representados com setas para cima.

$$
\begin{array}{lcl}
\langle S \uparrow x_0 \rangle & \leftarrow & \langle T \uparrow x_0 \rangle \\
\langle T \uparrow x_0 \rangle & \leftarrow & \langle B \uparrow x_0 \rangle \ (\langle B \uparrow x_1 \rangle \ [x_0 = 2 * x_0 + x_1])^* \\
\langle B \uparrow x_1 \rangle & \leftarrow & (0 \ [x_1 = 0]) \ / \ (1 \ [x_1 = 1])
\end{array}
$$

Figura 5 – APEG para números binários

Como um exemplo de como é o formalismo de APEG, a Figura 5 mostra um APEG que reconhece números binários e calcula o seu valor. Na terceira linha, cada uma das opções da escolha ordenada tem sua própria regra de avaliação, definindo que o valor da variável x1 é 0 (se a entrada é "0") ou 1 (se a entrada for "1"). Na segunda linha, o valor da variável x0 é inicialmente definido no primeiro uso do não-terminal B, sendo atualizado pela regra de avaliação $[x_0 = 2 * x_0 + x_1]$ a cada passo que o klenee for reconhecido. Nesse exemplo todos os atributos sintetizados representados com a seta direcional para cima, ou seja, todos esses atributos são valores de retorno.

2.3 Conclusão

Neste capítulo, foi apresentado as definições de PEG e APEG, assim como exemplos de como utilizá-los.

PEG é um reconhecedor da esquerda para a direita, também chamado de ordem prioritária, processando todo não-terminal a esquerda até consumir toda a entrada, a ordem prioritária elimina a ambiguidade comum em gramáticas como as gramáticas livre de contexto.

APEG estende PEG com a finalidade de estender a gramática original em tempo de análise de entrada, permitindo que linguagens extensíveis possam ser escritas no formalismo APEG.

3 Geradores de Analisadores Sintáticos de PEG

Rats! (GRIMM, 2006) e Mouse (REDZIEJOWSKI, 2009) são famosos geradores de analisador sintático para PEG.

Nesse capítulo será apresentado como são as caracteristicas de Rats! (Seção 3.1) e Mouse(Seção 3.2) e como essas ferramentas fazem a geração do analisador sintático para PEG.

3.1 Rats!

Rats! (GRIMM, 2006) é um gerador de analisador sintático para PEG, recebe uma gramática PEG como entrada e gera um analisador sintático escrito na linguagem Java. Rats! é uma ferramenta que implementa na prática um packrat parsing.

Packrat parsing é uma técnica para a geração eficiente de analisadores sintáticos descendente que executam retrocesso, armazena todos os resultados intermediários garantindo um desempenho em tempo linear mas por um grande custo de memória.(REDZIEJOWSKI, 2009)

Rats! permite a construção de gramáticas a partir da extensão de outras e modulares, oferecendo um ótimo nível de reutilização. Para fornecer reutilização de gramáticas, é possível importar módulos e estender suas regras de produção. A extensibilidade é realizada apenas estaticamente. (REIS; IORIO; BIGONHA, 2014a)

A Figura 6 descreve todos os operadores reconhecidos por Rats! juntamente com a ordem de precedência específica para cada operador.

A precedência especifica qual operador tem maior prioridade em relação ao outro. Assim os operadores de precedência 5 tem maior prioridade em relação aos operadores de precedência 1, ou seja, os operadores de precedência 5 são resolvidos primeiro do que os operadores de precedência 1.

Os operadores de Rats! segundo a Figura 6 são:

- ‘ ’ -> Especifica um literal;
- “ ” -> Especifica uma sequência de literais;
- [ ] -> Especifica uma classe de caracter;

Operator	Type	Prec.	Description
' '	Primary	5	Literal character
" "	Primary	5	Literal string
[ ]	Primary	5	Character class
.	Primary	5	Any character
{ }	Primary	5	Semantic action
(*e*)	Primary	5	Grouping
e?	Unary suffix	4	Option
*e**	Unary suffix	4	Zero-or-more
e+	Unary suffix	4	One-or-more
&*e*	Unary prefix	3	And-predicate
!*e*	Unary prefix	3	Not-predicate
id:*e*	Unary prefix	3	Binding
" ":*e*	Unary prefix	3	String match
e_1 e_2	Binary	2	Sequence
e_1 / e_2	Binary	1	Ordered choice

Figura 6 – Operadores em Rats!

- . -> Especifica quando não importa qual o é próximo caracter, então reconhece qualquer caracter a frente;
- (e) -> Especifica um grupo específico de caracteres;
- e? -> Especifica que a expressão é opcional;
- e* -> Especifica uma expressão que pode aparecer 0 ou mais vezes na linguagem;
- e+ -> Especifica uma expressão que deve aparecer pelo menor 1 vez ou pode aparecer mais vezes na linguagem;
- &e -> Especifica que se a expressão for reconhecida retorna verdadeiro mas retrocede a entrada;
- !e -> Especifica que se a expressão não for reconhecida retorna verdadedeiro;
- id:e -> Especifica que é obrigatório escrever o id para a expressão;
- “ ”: e -> Especifica se a string casa com a expressão;
- e1 e2 -> Especifica uma sequência de expressões;
- e1 / e2 -> Especifica escolhas ordenadas entre expressões;

A Figura 7 apresenta um exemplo de gramática para a linguagem L = { ab } escrita na sintaxe do Rats!.

```
public void s = 'a' 'b';
```

Figura 7 – Arquivo de texto de entrada para a ferramenta Rats

Essa gramática reconhece somente o caracter ‘a’ seguido de ‘b’, ou seja, a palavra ab. O analisador sintático gerado por Rats! para a gramática da Figura 7 é mostrado na Figura 8.

```
    // Todos os nao terminais recebem um ponteiro que ira
    // percorrer a entrada

  public Result ps(final int yyStart) throws IOException {

    int         yyC;
    int         yyIndex;
    Void        yyValue;
    ParseError  yyError = ParseError.DUMMY;

    // Alternative 1.
    //yyC recebe o caracter em que o ponteiro aponta
    yyC = character(yyStart);

    // se o ponteiro aponta para o caracter 'a', o ponteiro
    //passa a apontar para o proximo caracter da entrada.
    if ('a' = = yyC) {
      yyIndex = yyStart + 1;
      yyC = character(yyIndex);
      if ('b' = = yyC) {
        yyIndex = yyIndex + 1;
        yyValue = null;
        return new SemanticValue(yyValue, yyIndex, yyError);
      }
    }
    // Done.
    yyError = yyError.select("s expected", yyStart);
    return yyError;
  }
```

Figura 8 – Analisador sintático gerado automaticamente por Rats!

Cada não terminal, ou seja , cada função em Rats! recebe como parâmetro um ponteiro yyStar que aponta para cada string passada na entrada. A variável yyC é um ponteiro para caracter corrente da entrada. A variável yyValue contém o valor semântico atribuído ao lexema reconhecido.

A Entrada reconhecida pela gramática é: ‘ab’. O ponteiro apontará primeiro para ‘a’ e depois será incrementado passando a apontar para ‘b’.

Ao observar o analisador sintático gerado por Rats! na Figura 8.

Considere a seguinte gramática L = { w | w = cab } da Figura 10.

```
public void s = cc 'a' 'b';
public void cc = 'c'
```

Figura 9 – Arquivo de texto de entrada para a ferramenta Rats!

Rats! faz uma simplificação retirando todos os não terminais que geram terminais. Nessa gramática o não terminal cc será substituido como 'c'. O resultado final ficará:

```
public void s = 'c' 'a' 'b';
```

Figura 10 – Arquivo de texto de entrada para a ferramenta Rats!

Essa computação faz com que Rats! gere menos funções e simplifique toda a gramática.

Considere a seguinte gramática L = { w | w = cab ou dab } da Figura 11 :

```
public void s = cc 'a' 'b';
public void cc = ddcc;
public void ddcc = 'c' / 'd'
```

Figura 11 – Arquivo de texto de entrada para a ferramenta Rats!

Como o não terminal cc simplemesmente chama o não terminal ddcc, ao gerar o analisador sintático, Rats! faz com que cc tenha os valores $'c'/'d'$. Produzindo assim somente duas funções, reduzindo assim a quantidade de funções desnecessárias.

O analisador sintático gerado por Rats! não foi pensado para que seja legível ao usuário, sendo de difícil entendimento, mas Rats! tenta fazer com que o código produza o menor número de funções possíveis.

3.2 Mouse

Mouse (REDZIEJOWSKI, 2009) é um gerador de analisador sintático descendente recursivo escrito em Java, Mouse recebe uma gramática escrita em PEG e gera um analisador sintático para essa gramática em Java. O analisador sintático gerado pelo Mouse é mais fácil de ser entendido pelo usuário do que o analisador sintático gerado pelo Rats!.

A Figura 12 descreve os operadores reconhecidos por Mouse.

Os operadores de mouse segundo a Figura 12 são:

- " " -> Especifica uma sequência de literal;

expression	name	precedence
"s"	String Literal	5
[s]	Character Class	5
^[s]	Not Character Class	5
[c_1-c_2]	Character Range	5
_	Any Character	5
e?	Optional	4
e*	Iterate	4
e+	One or More	4
e_1*+e_2	Iterate Until	4
e_1++e_2	One or More Until	4
&e	And-Predicate	3
!e	Not-Predicate	3
$e_1 \ldots e_n$	Sequence	2
e_1 / ... / e_n	Choice	1

Figura 12 – Operadores em mouse

- [s] -> Especifica uma classe de um caracter;
- $\hat{}$[s] -> Especifica que não é uma classe específica de um caracter;
- [c1-c2] -> Especifica uma sequência com um valor máximo e mínimo;
- _ -> Especifica quando não importa qual o é próximo caracter, então reconhece qualquer caracter a frente;
- e? -> Especifica que a expressão é opcional;
- e* -> Especifica uma expressão que pode aparecer 0 ou mais vezes na linguagem;
- e+ -> Especifica uma expressão que deve aparecer pelo menor 1 vez ou pode aparecer mais vezes na linguagem;
- $e_1 * + e_2$ -> Especifica uma interação e_1 até e_2;
- e_1++e_2 -> Especifica uma expressão ou mais;
- &e -> Especifica que se a expressão for reconhecida retorna verdadeiro mas retrocede a entrada;
- !e -> Especifica que se a expressão não for reconhecida retorna verdadeiro;
- e1 e2 -> Especifica uma sequência de expressões;
- e1 / e2 -> Especifica escolhas ordenadas entre expressões;

Seja a Figura 13 um exemplo de gramática escrita na sintaxe de Mouse.

A gramática descrita acima primeiramente reconhece um número e em seguida adiciona 0 ou mais símbolos + seguido de outro número. Eis alguns exemplos de palavras aceitas pela reconhecedor a partir da gramática: 2+3, 3+4+5, 3+4+5+6.

```
Sum = Number ("+" Number)* !_ ;
Number = [0-9]+ ;
```

Figura 13 – Arquivo de texto de entrada para a ferramenta Mouse

Para essa gramática Mouse produz o seguinte analisador sintático da Figura 14.

```
//=====================================================================
//  Sum = Number ("+" Number)* !_ ;
//=====================================================================
private boolean Sum()
  {
    begin("Sum");
    if (!Number()) return reject();
    while (Sum_0());
    if (!aheadNot()) return reject();
    return accept();
  }

//---------------------------------------------------------------------
//  Sum_0 = "+" Number
//---------------------------------------------------------------------
private boolean Sum_0()
  {
    begin("");
    if (!next('+')) return rejectInner();
    if (!Number()) return rejectInner();
    return acceptInner();
  }

//=====================================================================
//  Number = [0-9]+ ;
//=====================================================================
private boolean Number()
  {
    begin("Number");
    if (!nextIn('0','9')) return reject();
    while (nextIn('0','9'));
    return accept();
  }
```

Figura 14 – Parser Gerado por Mouse

Mouse cria para cada não terminal uma função, o não terminal Sum possui uma expressão complexa, ou seja, um PEG dentro de outro PEG, possuindo uma sequência e dentro dessa sequência um klenee, nesses casos Mouse quebra a função em problemas menores produzindo uma subfunção.

Sum passa a ser:

```
Sum = Number Sum_0 !_ ;
Sum_0 = ("+" Number)*;
```

Mouse possui alguns métodos padrões:

- begin (): Marca qual a função atual. Subfunções não são marcadas.
- next (): Tenta casar um literal.

- accept () : Retorna True se a função faz o reconhecimento com sucesso da entrada.
- reject () : Retorna False se a função não faz o reconhecimento com sucesso da entrada.

Mouse tem como vantagem o fato de que a sua sintaxe e o analisador sintático gerado são mais legíveis, sendo de fácil entendimento para o usuário.

3.3 Conclusão

Rats! e Mouse são geradores de analisador para PEG, mas não oferecem suporte a extensibilidade, embora elas podem ser utilizadas para implementação de linguagens extensíveis.

APEG se diferencia de Rats! e Mouse por ser uma ferramenta para criação de linguagens extensíveis com suporte a extensibilidade. APEG é um primeiro estágio para criação de compiladores com suporte a extensibilidade.

4 Gerador de Analisador Sintático de APEG

O objetivo é criar uma ferramenta que processe uma gramática escrita na sintaxe de APEG e produz como resultado uma classe Java que implementa o analisador sintático *top down* correspondente à gramática de entrada. O analisador sintático produzido é implementado por meio de funções mutuamente recursivas, como usual para implementação de analisadores sintáticos *top down*. Optamos pela implementação em Java pela facilidade de codificação da solução e também como um primeiro experimento. Naturalmente as técnicas empregadas neste trabalho podem ser adaptadas para fazer com que a ferramenta produza código em outras linguagens.

Antes de começarmos a descrever a geração de código, vale atentar para algumas peculiaridades de APEG. Primeiramente APEG possui a noção de atributos e computações sobre esses atributos. O conjunto de operações possíveis a um atributo é fixa e pré-definida pela semântica de APEG, usualmente permitindo atribuições, operações aritméticas, lógicas e de comparação. Em particular há um atributo especial chamada atributo linguagem que contém uma definição da sintaxe corrente. Tal atributo também possui certas operações específicas, distintas das demais operações citadas.

Dado que os atributos podem ser definidos por computações, expressas na sintaxe de APEG, é necessário verificar que todas as operações realizadas entre todos os atributos sejam verificadas em relação a seus tipos, antes que o código do parser possa ser gerado. Nesse primeiro momento iremos nos preocupar apenas com erros de tipos que podem ser detectados de forma estática. Além da verificação de tipos de atributos, a ferramenta deve ainda verificar os usos de todos os símbolos de não terminais, reportando erros e avisos de forma apropriada.

A Figura 15 sumariza o processo de geração de código para um arquivo APEG. Tal como no processo de compilação usual o parser de APEG recebe como entrada uma gramática APEG e produz uma AST, na qual realizamos o processo de verificação de tipos e a construção da tabela de símbolos. A tabela de símbolos produzida é então usada no processo de geração de código.

A seguir apresenta-se uma descrição do sistema de tipos e do gerador de código para APEG desenvolvidos nesse trabalho.

4.1 Verificação de tipos

O sistema tem por objetivo determinar, em tempo de compilação, os seguintes erros de tipo:

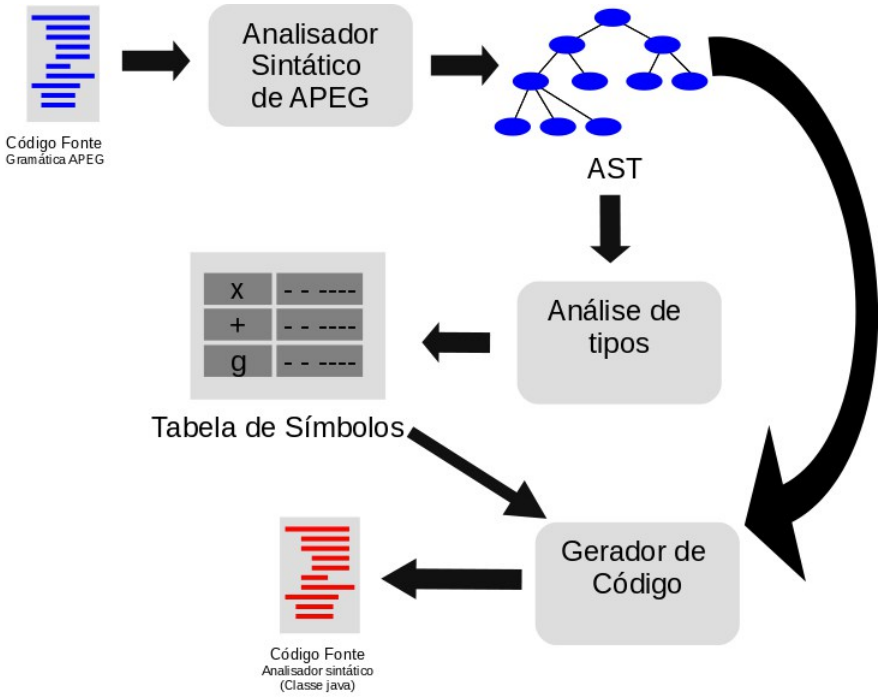

Figura 15 – Sequencia de ações para compilação de um arquivo fonte de APEG

- O uso de um terminal que não foi definido: O usuário pode se esquecer de prover a definição de um não terminal que é utilizado na definição de algum outro não terminal. Exemplo:

```
s: '0' a<0> b<> ;
a [int x] :  'c';
```

  Nessa gramática, o não terminal `b` não foi definido.

- Definição de não-terminal que nunca é utilizado: O usuário pode definir uma regra e se esquecer de utilizá-la, gerando um não terminal inalcançável a partir do símbolo de partida. Exemplo:

```
s: '0' a<0> ;
a [int x] :  'd';
c: ('0' / '1')*;
```

  Nesse exemplo a regra `c` é inalcançável a partir do símbolo de partida `s`.

- Chamada incorreta de um não terminal (aridade e tipo dos parâmetros): Ao chamar um não terminal o usuário pode passar mais argumentos ou menos argumentos do que esperado pelo não terminal, ou ainda passar argumentos de tipo que não é esperado pelo não terminal. Exemplo:

```
s: '0' a<'h'> c<0,1,2,3,4,5,6,7,8>;
```

```
a [int x] :  'd';
c: ('0' / '1')*;
```

Nesse exemplo a regra `a` espera um único parâmetro de tipo inteiro, mas na regra `s` ela é chamada com um argumento de tipo carácter. A regra `c` não recebe e nem retorna parâmetros de quaisquer tipos mas é chamada em `s` com vários argumentos.

- Valores passados no lugar de parâmetros sintetizados:

```
s: '0' a<0> ;
a retuns [int x] :  'd';
```

Nesse exemplo como a somente possui uma variável de retorno este não pode ser um valor e sim uma variável que receba esse retorno.

- Erros de tipos em expressões: Todos os usos de operadores em APEG dever se considerados bem tipados se, e somente se, aplicados a parâmetros de mesmo tipo.

```
s [int a, float b]: a + b ;
```

Nessa gramática pode-se observar que os dois tipos não são compatíveis, e portanto essa gramática não está bem tipada.

A implementação do verificador de tipos foi feita usando o padrão de projetos *visitor* que permite percorrer a AST produzida pelo analisador sintático. O *visitor* utilizado não fixa a ordem de visita dos nós permitindo evitar certos percursos que não são necessários em um dado momento.

A verificação de tipos foi implementando em dois *visitors*. O primeiro, a saber `BuildRuleEnvironmetVisitor.java`, se concentra em coletar nomes e tipos de não terminais e criar uma tabela de símbolos com essas informações. O segundo, a saber `VerifyVisitor.java` usa a tabela símbolos construída pelo primeiro e se concentra em verificar o corpo de cada não terminal, possivelmente adicionando mais informações sobre variáveis locais à tabela de símbolos.

A razão para se realizar dois percursos na árvore é que ao verificar o corpo de um não terminal, podemos encontrar uma referência para um não terminal que ainda não foi definido até o momento. Deste modo não poderíamos determinar se o uso desse não-terminal está correto. No entanto, se já conhecemos todos os não terminais de antemão, podemos determinar se o uso de qualquer não terminal está correto ou não, no momento em que o encontramos. Note que essa estratégia permite verificar definições recursivas de não-terminais de forma natural.

4.1.1 A Estrutura da Tabela de Símbolos

A tabela de símbolos contém informações sobre o tipo da regra, assim como o nome e o tipo de cada variável declarada em seus parâmetros e se essa variável é um parâmetro herdado ou sintetizado. O tipo de uma regra pode ser interpretado como o tipo de uma função em tuplas ($(h_1, ...h_n) \rightarrow (s_1, ..., s_m)$), onde $(h_1, ...h_n)$ representam os argumentos herdados e $(s_1, ..., s_m)$ representa os argumentos sintetizados.

Representamos o tipo de uma regra por meio de uma instância da classe `NTType.java`, localizada no pacote `apeg.parse.ast.visitor.Environments`, que representa o tipo de um não terminal com um arranjo de objetos do tipo `TypeNode`. Os tipos no arranjo são organizados na mesma ordem que ocorrem na assinatura de tipo do não terminal o que significa que todos os tipos de parâmetros herdados estão antes de todos os tipos de parâmetros sintetizados. As informações sobre o tipo de variáveis locais de um não terminal são representadas por instâncias da classe `VarType`, no mesmo pacote. Cada objeto dessa classe carrega informa o tipo, direção do atributo (herdado,sintetizado ou local) e um código de acesso para a variável.

Finalmente cada linha da tabela de símbolos é representada por um mapeamento de um nome de não-terminal para uma instância da classe `NTInfo`, que abriga uma instância da classe `NTType`, o tipo do não terminal, e uma tabela de mapeamento de nomes de variáveis para instâncias da classe `VarType` que contém as informações sobre as variáveis locais. Para exemplificar essa estrutura, considere a gramática da Figura 16 para a simples linguagem regular $L = \{0^n1^m | n, m \geq 0\}$. Após executarmos o primeiro *visitor*, `BuildRuleEnvironmetVisitor`, sobre essa gramática obtemos a tabela de símbolos como vista na Figura 17. Note que apenas os tipos dos parâmetros foram adicionados à tabela. De fato o `BuildRuleEnvironmetVisitor` apenas visita os nós referentes a definição de cada regra, ignorando qualquer outro nó da AST.

```
s:  a<n> b<m> ;
a returns [int n] : {x = 0;}({x = x+1;} '0')*{n = x;} ;
b returns [int m]:  {y = 0;} ('1' {y = y + 1;})* {m = y;};
```

Figura 16 – Gramática APEG para 0^n1^m com $n, m \geq 0$

Figura 17 – Tabela de símbolos produzida pelo BuildRuleEnvironmetVisitor

		Tabela de variáveis locais			
Nome da regra	Tipo da Regra	Nome	Tipo	H/S	Cod. Acesso.
s	$() \rightarrow ()$				
a	$() \rightarrow (Int)$	n	Int	S	0
b	$() \rightarrow (Int)$	m	Int	S	0

Note que o tipo de s é representado como $() \rightarrow ()$, evidenciando que essa regra não recebe nenhum parâmetro como entrada e não produz qualquer valor como resultado. A coluna Tipo informa o tipo de uma variável local e a coluna H/S informa se a variável é atributo herdado (H), sintetizado (S) ou uma variável local (nem herdada e nem sintetizada). A coluna `Cod. Acesso` define o código de acesso da variável. Esse código é obtido enumerando-se cada declaração de variável dentro da regra, a medida que as encontramos. Essa informação será posteriormente utilizada para mapear a variável no contexto de valores da regra.

Após a execução do primeiro *visitor*, a tabela obtida é passada ao segundo *visitor*, `VerifyVisitor`, que realiza a verificação de tipos dentro do corpo da regra. Ao ser criado o `VerifyVisitor` produz um lista dos nomes de todos os não terminais na tabela. A cada uso de um não terminal seu respectivo nome é removido da lista, caso ele esteja presente. Ao final do processo, se ainda restarem nomes de não terminais na lista, sabemos que esses são nomes de não terminais que nunca foram usados.

Em seguida verificam-se as variáveis locais definidas dentro de cada regra e as expressões. A cada atribuição testamos se a variável foi definida é a tabela de símbolos se tornará então como a tabela vista na Figura 18. Para fins de comodidade, nessa etapa, uma definição de uma variável pode funcionar como uma declaração, desde que a variável ainda não tenha sido declarada e que seus usos permaneçam consistentes daquele ponto em diante.

Figura 18 – Tabela de símbolos estendida pelo VerifyVisitor

		Tabela de variáveis locais			
Nome da regra	Tipo da Regra	Nome	Tipo	H/S	Cod. Acesso.
s	$() \rightarrow ()$				
a	$() \rightarrow (Int)$	n x	Int Int	S Local	0 1
b	$() \rightarrow (Int)$	m y	Int Int	S Local	0 1

4.1.2 Verificação de Tipos para Expressões

Ao encontrar uma atribuição $v = e$ na qual v é um identificador e e é uma expressão qualquer o `VerifyVisitor` infere o tipo da expressão da expressão e e verifica se esse tipo é compatível com o tipo da variável registrado na tabela de símbolos, referente ao contexto da regra sendo correntemente verificada.

A implementação da inferência de tipos foi feita por um caminhamento *pós-order* na sub-árvore de expressões. Ao visitar nós folhas os tipos de variáveis e literais são

empilhados. Ao visitar nós intermediários, tipicamente operadores, os n últimos tipos são desempilhados, sendo n a aridade do operador, e o tipo resultante da operação é empilhado. Quando um erro de tipo é descoberto, um tipo `TyError` especial, que representa erro, é empilhado como resultado e o erro é registrado em uma lista de erros. Esse mecanismos permite que a verificação de tipos prossiga e e encontre todos os erros de tipos.

Para que os operadores aritméticos sejam considerados bem tipados, ambos os argumentos devem ter o mesmo tipo e portanto expressões como `2 + 3.0` são consideradas mal tipadas. A princípio a escolha pode parecer muito restritiva, no entanto não temos intenção que APEG seja utilizada para processamento numérico intenso. Todavia, esse comportamento pode ser facilmente modificado dado que as regras para tipagem de operadores são passadas como parâmetro para o emphvisitor e são implementadas na forma de uma tabela, como a Tabela 19

Figura 19 – Tabela de tipos de operadores

Operador	Parâmetros		Resultado
$\oplus$	Int	Int	Int
$\oplus$	Float	Float	Float
$\odot$	Bool	Bool	Bool
$\boxplus$	Int	Int	Bool
$\boxplus$	Float	Float	Bool
$\boxplus$	Char	Char	Bool

Na Tabela 19 o símbolo $\oplus$ representa os operadores aritméticos `+,-,* e /`, o símbolo $\odot$ representa os operadores `& e |`, o símbolo $\boxplus$ representa os operadores relacionais `>,< e ==`.

4.2 Geração de código

Uma vez que o código não contenha erros de tipos, a tabela de símbolos produzida na etapa anterior e a AST são submetidas a etapa de geração de código. Neste trabalho nos concentramos em gerar código para a linguagem Java. Isto significa que a saída do gerador de código será um arquivo de texto contendo uma classe Java que implementa o analisador sintático para gramática submetida. A estratégia empregada na etapa de geração de código é muito similar ao algoritmo para gerar um parser descendente recursivo a partir de uma GLC.

No entanto, ao contrário de GLCs, as alternativas em PEGs são determinísticas e devem ser realizadas na ordem em que são listadas e na ordem em que são especificadas. Adicionalmente quando é encontrado uma falha uma PEG se comporta como um analisador sintático descendente recursivo com retrocesso. Logo mecanismos para controlar os

retrocessos e sincronizar e entrada de dados de forma apropriada devem ser empregados de modo a garantir o comportamento correto do reconhecedor gerado.

A idéia principal dessa abordagem é compilar cada definição de um não terminal para uma função. Cada ocorrência de um terminal gera uma chamada de um *match* para uma cadeia de caracteres que descreve aquele terminal. Cada uso de um não terminal gera uma chamada a função que implementa aquele não terminal.

Para simplificar o código do reconhecedor gerado, e ao mesmo tempo controlar os retrocessos, optamos por criar uma biblioteca de classes com as funcionalidades básicas comum a todos os reconhecedores sintáticos. Essa biblioteca deve ser importada em cada reconhecedor sintático gerado e pode ser distribuída como pacote JAR separado da ferramenta principal. Chamamos essa biblioteca de RTS [1] Uma descrição detalhada da RTS e todos os seus componentes seria por demais tediosa e portanto apenas apresentaremos os conceitos básicos da RTS e omitiremos a implementação dos métodos relacionados a ela.

A RTS é formada pelas classes que descrevem a árvore de derivação sintática, uma classe base `BaseParser` que serve como base para os reconhecedores sintáticos gerados e uma classe que implementa um fluxo de dados com suporte a retrocesso.

A classe `BaseParser`, gerencia o estado do reconhecedor e o atualiza de forma apropriada. Há um conjunto de três pilhas, a saber a pilha de símbolos (ou de terminais), uma pilha de marcas e uma pilha de nome de regras (ou não terminais). A pilha de símbolos contém a frente da AST correntemente sendo montada, a pilha de marcas é usada para determinar em qual não terminal uma marca foi criada e a pilha de regras contém em seu topo a regra correntemente sendo processada. Adicionalmente a RTS também registra o estado de sucesso corrente do reconhecedor como sendo sucesso ou fracasso. Chamamos esse estado de `Ok` e seu valor é um *booleano*.

A Figura 20 apresenta um esquema do estado do `BaseParser`.

A principais operações do `BaseParser` e que estarão no código gerado são:

- startRule(String ruleName) : Empilha o nome da regra na pilha de regras.
- success(): Usado para informar que a regra terminou com sucesso. Causa a remoção do topo da pilha de regras e a consequente criação do ramo da AST associado a regra recém terminada.
- done(): Seta o `Ok` para verdadeiro.
- fail(): Usado para informar que a regra falhou. Seta o `Ok` para falso, e descarta todos os símbolos desde o início da regra corrente. Retrai o ramo referente à regra corrente

[1] do inglês *Run Time System*, Sistema de Tempo de execução. Esse nome foi dado pois essa biblioteca tem, de certa forma, o mesmo papel de uma RTS em um programa compilado para código objeto.

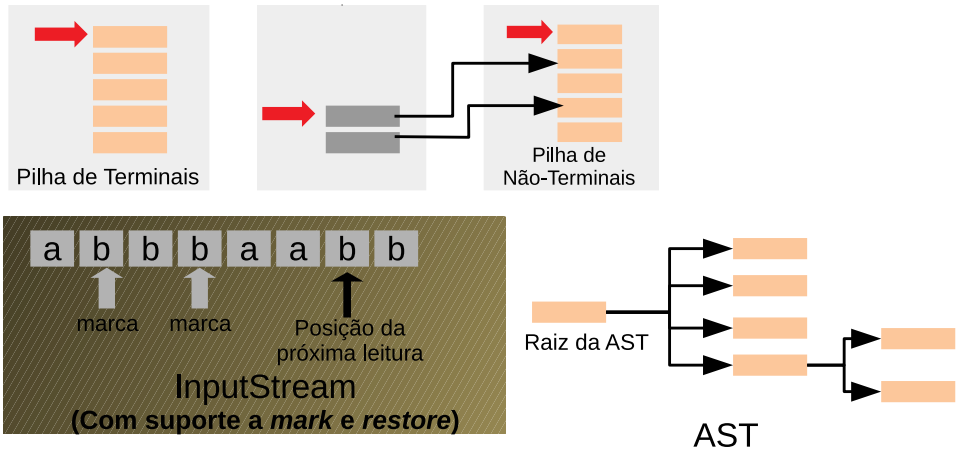

Figura 20 – Esquema dos componente classe base para os analisadores sintáticos

Fonte: Autor

da AST.

- failure(): Se comporta como fail(), exceto que e descarta todos os símbolos empilhados desde a última marca.
- undo(): Descarta todos os símbolos empilhados desde a última marca e recria a última marca feita. Mantém o o estado de `Ok` inalterado.
- mkBacktracPoint(): Cria um ponto de *backtracking*, registrando a posição do cursor no fluxo de entrada e o estado corrente do `BaseParser`.
- dismissBacktracPoint(): Dispensa e o último ponto de *backtracking* criado.
- restore(): restaura o estado do `BaseParser` para o último ponto de *backtracking* criado, dispensando em seguida.
- isOk(): Retorna o estado de `Ok`. Informa se última ação de reconhecimento terminou em sucesso ou fracasso.
- match(String s): Tenta casar s com o prefixo da entrada, setando o valor de `Ok` para verdadeiro caso obtenha sucesso ou para falso caso contrário.

Essas operações podem ser usadas para codificar reconhecedores sintáticos descendentes recursivos. O processo de construção da AST do programa sendo reconhecido é gerenciado automaticamente pelo `BaseParser`.

Para compilar os atributos precisa-se fazer com que as funções recebam, como parâmetros, os valores dos atributos. Adicionalmente a solução adotada neste momento precisa levar em consideração o fato de que APEG deve suportar extensões de regras.

Uma forma de criar uma solução mais ortogonal é representar todos os atributos de uma mesma regra como um vetor de valores *Object*, a esse vetor dá-se o nome de ambiente de valores. Cada variável deve então ser compilada para o seu respectivo endereço nesse vetor, o que já é feito pelo *visitor* de verificação de tipo quando ele calcula o código de acesso de uma variável.

A passagem de parâmetros é feita do seguinte modo: A regra chamadora cria o ambiente de valores da regra chamada, o inicializa e por fim chama a regra passando o ambiente de valores. Ao usar um valor da ambiente de valores o código da regra chamada realiza o *casting* do valor para o tipo apropriado.

Ao retornar da chamada, a regra chamadora deve copiar os valores do vetor correspondente aos parâmetros sintetizados do ambiente da regra chamada para as respectivas variáveis em seu próprio ambiente.

Finalmente para lidar com as situações de retrocessos, onde potencialmente pode-se fazer necessário descartar alterações previamente feitas em variáveis, encapsulou-se o ambiente de valores em um classe que, na verdade, gerencia uma pilha de ambientes de valores. A API da classe permite iniciar um modo temporário, que basicamente empilha uma cópia do ambiente atual na topo da pilha.

4.3 Código Gerado

O código gerado sempre iniciará da seguinte forma:

```
public class SimpleTest extends StateFullBaseParser{

 public SimpleTest(String fname) {
 super(fname);
 startRule("root");
 }
 .
 .
 .
}
```

Figura 21 – Código base gerado para toda gramática de entrada na ferramenta APEG

O código estende StateFullBaseParser que implementa as principais funções que fazem a análise da gramática de entrada. Os três pontos indicam que mais código será adicionado dependendo da gramática de entrada.

Para a geração de código diferentes decisões devem ser tomadas dependendo da expressão PEG a ser visitada. As possíveis expressões PEG e como são resolvidas são:

- **Choice**

A estratégia utilizada é marcar o inicio da função com mkBacktracPoint, se o match de 'a' deixar a máquina no estado true retorna-se sucesso, senão restaura e faz um novo mkBacktracPoint e tenta fazer o match de 'b', se a regra terminar com sucesso retorna sucesso, senão falha.

Abaixo está o código gerado para a gramática s: 'a' / 'b'.

```
public PegResult s(AttrEnvironment env){
  startRule("s");
  mkBacktracPoint();            // Marca-se a entrada
  match("a");                   //Tenta reconhecer o caracter 'a'
                                //retornando true ou false
  if(isOk()){ return success();}//Se o caracter 'a' foi reconhecido
                                // retorno sucesso
  restore();                    // caso contrario restauro a entrada
                                // retirando a marca que tinha sido
                                //feita inicialmente
  mkBacktracPoint();            // Marca-se a entrada novamente para
                                // a proxima escolha
  match("b");
  if(isOk()){ return success();}
  if(isOk()){return success();} else{return fail();}
 }
```

- **Sequence**

A estratégia é tentar o match de 'a' e somente se a máquina estiver em true tentar o match de 'b'.

Abaixo está o código gerado para a gramática s: 'a' 'b'

```
public PegResult s(AttrEnvironment env){
 startRule("s");
 match("a");                     // Tenta reconhecer 'a'
 if(isOk()){                     // Se o reconhecimento foi feito
                                 // com sucesso a proxima entrada
   match("b");                   // sera testada
 }
 if(isOk()){return success();} else{return fail();}
 }
```

- **Klenne**

A estrategia é fazer um do-while para enquanto o match de 'ab' estiver retornando verdadeiro e dar um mkbacktraking no inicio da função. No momento que para de reconhecer a entrada, tem-se que restaurar a entrada, se tudo deu certo tiro a marca. No final de tudo mesmo se não tiver dado match tem-se que dar um done.

Abaixo está o código gerado para a gramática s: 'ab'*'

```
public PegResults(AttrEnvironment env){
 startRule("s");
 do{
  mkBacktracPoint();       // Marca-se a entrada
  env.tempMode();
  match("a");              // Tenta fazer o reconhecimento da entrada
  if(!isOk()){             // Se a entrada nao for reconhecida
    restore();             // restauro a entrada e desmaco o ponto
    env.revokeChanges();
  }else{
    dismissBacktracPoint();
    env.consolidateTemp()
  }
 }while(isOk());
 env.endTempMode();
 done();                 //Independente se teve algum reconhecimento
                         // ou nao, a maquina deve estar no estado
                         // verdadeiro.
 if(isOk()){
   return success();
 } else{return fail();}
}
```

- **Plus Klenne**

  Funciona da mesma forma que o Klenne mas antes do do-while é obrigatório que um match seja satisfeito.

  Abaixo está o código gerado para a gramática s: 'ab'+

```
public PegResult s(AttrEnvironment env){
 startRule("s");
 match("ab");                  // ab deve ser reconhecido pelo
 if(!isOk()){                  // pelo menos uma vez, se nao for
   return fail();              // reconhecido, retorna-se falha.
 }
 do{                           // As outras vezes sao totalmente
   mkBacktracPoint();          // opcionais.
   env.tempMode();
   match("ab");
   if(!isOk()){
     restore();
     env.revokeChanges();
   }else{
     dismissBacktracPoint();
     env.consolidateTemp();
   }
 }while(isOk());
 env.endTempMode();
```

```
 done();
 if(isOk()){
   return success();
 }else{
   return fail();
 }
}
```

- **Not**

  A estratégia é marcar a entrada com mkBacktracPoint, se o match ocorrer dou um restaure e retorno fail. Senão dismis e done.

  Abaixo está o código gerado para a gramática s: !('ab')

```
public PegResult s(AttrEnvironment env){
 startRule("s");
 mkBacktracPoint();                  // Marca-se a entrada
 match("ab");                        // Testando o reconhecimento
 if(isOk()){                         // Se o reconhecimento retornou
   restore();                        // sucesso, a entrada deve ser
                                     // restaurada, e uma falha deve
   fail();                           // ser retornada
 }else{                              // Senao desmarca o ponto de
   dismissBacktracPoint();           // entrada e faz com que a
   done();                           // maquina va para o estado
 }                                   // verdadeiro
 if(isOk()){return success();} else{return fail();}
}
```

- **And**

  A estratégia é marcar a entrada com mkBacktracPoint e tentar casar 'a' e então dar um restore, se a máquina estiver no estado true retorno sucesso, senão retorno falha.

  Abaixo está o código gerado para a gramática s: & 'a'

```
public PegResult s(AttrEnvironment env){
  startRule("s");
  mkBacktracPoint();                // Marca-se a entrada
  match("ab");                      // Tenta fazer o reconhecimento
  restore();                        // Restaura e desmarca a entrada
  if(isOk()){return success();} else{return fail();}
}
```

- **Optional**

  A estratégia é marcar a entrada com mkBacktracPoint e tentar um match de b, se a máquina esta no estado true a marca é tirada, senão restauro e dou done, se tudo

estiver ok tento dar match em 'a', se ao final tudo estiver certo retorno sucesso senao retorno falha.

Abaixo está o código gerado para a gramática s: 'a' ?'b'

```
public PegResult s(AttrEnvironment env){
  startRule("s");
  mkBacktracPoint();                  // Marca-se a entrada
  match("a");                         // Tenta fazer o reconhecimento
  if(isOk()){ dismissBacktracPoint();// Se reconheceu demarca-se
                                      // o ponto de entrada
  }else{ restore(); done(); }         // Senao restaura a entrada e
                                      // faz com a que a maquina va
                                      // o estado verdadeiro
  if(isOk()){                         // sempre prosseguindo com o
    match("b");                       // reconhecimento.
  }
  if(isOk()){return success();} else{return fail();}
}
```

4.4 Conclusão

Neste capítulo todas as estratégias para a implementação da ferramenta APEG foram explicadas, assim como cada código gerado para cada APEG específico.

A ferramenta APEG recebe como entrada uma gramática APEG que será analisada pelo analisador sintático de APEG que consequentemente gerará uma AST. Essa AST então será percorrida e analisada para a geração correta do analisador sintático para APEG.

5 Conclusão e Trabalhos Futuros

Nesse trabalho foi desenvolvido o analisador sintático de PEG com atributos. Primeiramente a ferramenta recebe de entrada uma gramática APEG que será analisada pelo analisador sintático de APEG desenvolvido por Reis et. all. Esse analisador gerará uma AST que será percorrida com o auxílio de visitors específicos para cada situação.

Inicialmente foi desenvolvido um sistema de tipos para a AST gerada, esse sistema irá detectar todo o tipo de erro que a gramática de entrada possua, e então mostrará ao usuário todos os erros coletados, assim como possíveis warnings. Se a gramática de entrada possuir algum erro, a geração de código para o analisador sintático não será realizada.

A gramática de entrada não possuindo nenhum erro, uma tabela de símbolos com os tipos de cada não terminal será contruída e ajudará em todo o processo de codificação da ferramenta.

Com a união da AST e da tabela de símbolos, foi possível a implementação do analisador sintático para PEG com atributos. Inicialmente todos os PEGs podem ser compilados, assim como os seus atributos herdados e sintetizados.

5.1 Trabalhos Futuros

Como dito anteriormente, esse trabalho realizou a implementação da ferramenta PEG com atributos, ou seja, a adaptabilidade de APEG não foi implementada. Como trabalhos futuros teremos a implementação do interpretador que dê suporte a adaptabilidade de APEG. Modificações na ferramenta serão necessárias para uma ferramenta mais elegante e eficiente e assim como provas e testes de como o compilador está correto.

Referências

ERDWEG, S. et al. Sugarj: library-based syntactic language extensibility. In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. New York, NY, USA: ACM, 2011. (OOPSLA '11), p. 391–406. ISBN 978-1-4503-0940-0. Disponível em: <http://doi.acm.org/10.1145/2048066.2048099>. Citado na página 14.

FORD, B. Parsing expression grammars: a recognition-based syntactic foundation. In: . [S.l.]: ACM, 2004. p. 111–122. Citado 3 vezes nas páginas 12, 15 e 17.

GRIMM, R. Better extensibility through modular syntax. In: . [S.l.]: ACM, 2006. p. 38–51. Citado na página 19.

JOHNSON, S. C. Yacc: Yet another compiler compiler. In: . [S.l.]: Holt, Rinehart, and Winston, 1979. v. 2, p. 353–387. Citado na página 12.

REDZIEJOWSKI, R. R. Mouse: From parsing expressions to a practical parser. In: . [S.l.]: L.Czaja and M.Szczuka„ 2009. v. 2, p. 514–525. Citado 2 vezes nas páginas 19 e 22.

REIS, L. V. et al. The formalization and implementation of adaptable parsing expression grammars. *Science of Computer Programming*, v. 96, Part 2, p. 191 – 210, 2014. ISSN 0167-6423. Selected and extended papers of the Brazilian Symposium on Programming Languages 2012 (SBLP 2012). Disponível em: <http://www.sciencedirect.com/science/article/pii/S0167642314000872>. Citado na página 12.

REIS, L. V. S. et al. Adaptable parsing expression grammars. In: . [S.l.]: Springer Berlin Heidelberg, 2012. v. 7554, p. 72–86. Citado 2 vezes nas páginas 12 e 15.

REIS, L. V. S.; IORIO, V. O. D.; BIGONHA, R. S. Defining the syntax of extensible languages. In: . [S.l.]: ACM, 2014. p. 1570–1576. Citado 4 vezes nas páginas 12, 15, 17 e 19.

REIS, L. V. S.; IORIO, V. O. D.; BIGONHA, R. S. A mixed approach for building extensible parsers. In: . [S.l.]: Springer International Publishing, 2014. v. 8771, p. 1–15. Citado na página 12.

WILSON, G. V. Extensible programming for the 21st century. ACM, v. 2, p. 48–57, 2004. Citado na página 12.

TERMO DE RESPONSABILIDADE

Eu, Deise Kelley Silva, declaro que o texto do trabalho de conclusão de curso intitulado "Um Gerador de analisador sintático para adaptable parsing expression grammars" é de minha inteira responsabilidade e que não há utilização de texto, material fotográfico, código fonte de programa ou qualquer outro material pertencente a terceiros sem as devidas referências ou consentimento dos respectivos autores.

João Monlevade, 23 de maio de 2019

Deise Kelley Silva
Assinatura do aluno

DECLARAÇÃO DE CONFORMIDADE

Certifico que o(a) aluno(a) **Deise Kelley Silva**, autor do trabalho de conclusão de curso intitulado "**Um Gerador de Analisador Sintático para Adaptable Parsing Expression Grammar**" efetuou as correções sugeridas pela banca examinadora e que estou de acordo com a versão final do trabalho.

João Monlevade, 13 de maio de 2019.

Elton M. Cardoso
Professor (a) Orientador (a)