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

DANIEL FERREIRA PINTO

# IMPLEMENTAÇÃO ESCALÁVEL DO ANALISADOR SINTÁTICO DE APEG

João Monlevade

2017

DANIEL FERREIRA PINTO

# IMPLEMENTAÇÃO ESCALÁVEL DO ANALISADOR SINTÁTICO DE APEG

Monografia apresentada ao curso de Sistemas de Informação 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: Leonardo Vieira dos Santos Reis

João Monlevade

2017

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E APLICADAS
COLEGIADO DO CURSO DE SISTEMAS DE INFORMAÇÃO

**ANEXO III – Termo de Responsabilidade**

## TERMO DE RESPONSABILIDADE

Eu, Daniel Ferreira Pinto, declaro que o texto do trabalho de conclusão de curso intitulado *"Implementação Escalável do Analisador Sintático de APEG"* 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, 24 de julho de 2017.

Daniel Ferreira Pinto
Assinatura do aluno

**ATA DE DEFESA**

Aos vinte e um dias do mês de julho de dois mil de dezessete, às dezenove horas horas e vinte minutos, na sala H102 do Instituto de Ciências Exatas e Aplicadas, foi realizada a defesa de Monografia pela aluna **Daniel Ferreira Pinto**, sendo a Comissão Examinadora constituída pelos professores: Prof. Dr. Leonardo Vieira dos Santos Reis, Prof. Me. Elton Máximo Cardoso, Prof. Dr. Gleiph Ghiotto Lima de Menezes.

O candidato apresentou a monografia intitulada: *"Implementação Escalável do Analisador Sintático de APEG"*. A comissão examinadora deliberou por unanimidade, pela aprovação do candidato, com nota 7,0 (Sete pontos), concedendo-lhe o prazo de 15 dias para incorporação das alterações sugeridas ao texto final.

Na forma regulamentar, foi lavrada a presente ata que é assinada pelos membros da Comissão Examinadora e pelo graduando.

João Monlevade, 21 de julho de 2017.

Leonardo Vieira dos Santos Reis

Prof. Dr. Leonardo Vieira dos Santos Reis
Professor Orientador/Presidente

Elton M. Cardoso

Prof. Me. Elton Máximo Cardoso
Professor Convidado

Gleiph Ghiotto Lima de Menezes

Prof. Dr. Gleiph Ghiotto Lima de Menezes
Professor Convidado

Daniel Ferreira Pinto

Daniel Ferreira Pinto
Graduando

**FOLHA DE APROVAÇÃO DA BANCA EXAMINADORA**

***Implementação Escalável do Analisador Sintático de APEG***

**Daniel Ferreira Pinto**

**Monografia apresentada ao Instituto de Ciências Exatas e Aplicadas da Universidade Federal de Ouro Preto como requisito parcial da disciplina CSI499 – Trabalho de Conclusão de Curso II do curso de Bacharelado em Sistemas de Informação e aprovada pela Banca Examinadora abaixo assinada:**

Prof. Dr. Leonardo Vieira dos Santos Reis
Departamento de Computação e Sistemas - DECSI

Prof. Me. Elton Máximo Cardoso
Departamento de Computação e Sistemas - DECSI

Prof. Dr. Gleiph Ghiotto Lima de Menezes
Departamento de Computação e Sistemas - DECSI

João Monlevade, 21 de julho de 2017

*Dedico este trabalho aos meus pais, Lázaro e Tânia e a minha irmã Jaqueline por sempre estarem me apoiando, sem eles nada disso teria acontecido.*

# Agradecimentos

Agradeço primeiro a Deus por me conceder o dom da vida e ter me dado saúde para conseguir alcançar essa conquista. Agradeço a minha família pelo apoio, carinho, incentivo que me deram por todos esses anos de luta, sem eles eu não teria chegado até aqui, meu amor por vocês não pode ser medido. Aos meus amigos de longa data Cristian, Leandro e toda a Galera do Chápeu, pela amizade e companheirismo mesmo na distância, a alegria de vocês foi muito importante. Aos novos amigos que fiz durante o curso, tanto os de sala de aula quanto os com quem dividi moradia, foi um prazer conhecer a todos. A Luciane, que esteve presente todos esses anos comigo, desde o ínicio quando vi meu nome como aprovado até chegar aqui, no final dessa jornada.

Quero deixar um agradecimento especial ao meu orientador Dr. Leonardo Reis, pela paciência, pela ajuda e assistência que me deu durante o tempo em que fiz esse trabalho, foi um prazer ser orientado pelo senhor. Aos professores que tive durante o curso, por dividirem seu conhecimento e ensinar além de conteúdo didático como ser um profissional melhor. Agradeço a todos que fizeram parte dessa caminhada contruibuindo para que eu conseguisse chegar até o final.

*"Ainda que eu ande*
*pelo vale da sombra da morte,*
*não temerei mal algum,*
*pois tu estás comigo..."*
*(Sl 23.4)*

# Resumo

Neste trabalho, foi implementado um analisador sintático escalável para Adaptable Parsing Expression Grammar. A metodologia utilizada foi baseada na metodologia proposta por Ryu (RYU, 2015) na qual foi usada a ferramenta Rats! para especificação sintática, pois essa é capaz de separar partes da gramática em módulos. Foi acrescentado um processo automatizado de testes para validar as especificações feita com a ferramenta. Para a automatização dos testes foi utilizado o JUnit.

**Palavras-chaves**: APEG; modularidade; linguagens de programação, analisador sintático.

# Abstract

A scalable syntax analyzer for Adaptable Parsing Expression Grammar was developed in this project. The employed methodology is based on the one suggested by Ryu (RYU, 2015), which uses the Rats! tool to specify the syntax since it is able to separate parts of the grammar in modules. An automatic testing process was performed to validate the specification. The JUnit was used to automatize the tests.

**Key-words**: APEG; modularity; programming languages, parsing generator.

# Lista de ilustrações

Figura 1 – Linguagem escrita em PEG . . . . . 24
Figura 2 – Exemplo de processamento de uma entrada em PEG . . . . . 24
Figura 3 – Sintaxe da linguagem extensível *uSugar* . . . . . 26
Figura 4 – Exemplo de programa em *uSugar* . . . . . 27
Figura 5 – Módulo Expressão . . . . . 30
Figura 6 – Módulo Soma . . . . . 30
Figura 7 – Módulo Produto . . . . . 30
Figura 8 – Módulo Valor . . . . . 30
Figura 9 – Diagrama de pré-requisitos dos módulos da gramática de APEG . . . . 31

# Lista de abreviaturas e siglas

| | |
|---|---|
| PEG | Parsing Expression Grammar |
| APEG | Adaptable Parsing Expression Grammar |
| AST | Abstract Syntax Tree |

# Sumário


1 INTRODUÇÃO . . . . . 21
1.1 Problema . . . . . 21
1.2 Objetivo . . . . . 21
1.3 Metodologia . . . . . 22
1.4 Contribuições . . . . . 22
1.5 Organização da Monografia . . . . . 22

2 ADAPTABLE PARSING EXPRESSION GRAMMAR . . . . . 23
2.1 Parsing Expression Grammar . . . . . 23
2.2 Adaptable Parsing Expression Grammar . . . . . 25
2.3 Conclusão . . . . . 27

3 FERRAMENTAS PARA ESPEFICICAÇÃO SINTÁTICA . . . . . 29
3.1 Rats! . . . . . 29
3.2 ASTGen . . . . . 32
3.3 Especificação de Testes . . . . . 33
3.4 Conclusão . . . . . 33

4 CONCLUSÃO . . . . . 35
4.1 Trabalhos futuros . . . . . 35

REFERÊNCIAS . . . . . 37

# 1 Introdução

No projeto de novas linguagens há mudanças frequentes em sua sintaxe. Tais modificações na sintaxe acarretam a modificação de suas regras gramaticais, as quais são, de difícil adaptação dependendo da complexidade da gramática e o número de regras que ela possui. Considerando que as demais fases de um projeto de um compilador dependem do resultado produzido pelo analisador sintático, tais mudanças na sintaxe refletem também na Abstract Syntax Tree (AST) e, por consequências, nas fases posteriores.

Assim, um projeto de analisador sintático que facilite modificações tem relevância, sendo objetivo de estudo do trabalho publicado por Ryu. Ryu (2015) aborda como as lições aprendidas na implementação de Fortress são aplicadas no desenvolvimento de JavaScript. São usadas ferramentas que permitem definir modularmente a sintaxe da linguagem e um gerador automático AST.

*Adaptable Parsing Expression Grammar* (APEG) (REIS et al., 2012) é um formalismo ainda em desenvolvimento e, por isso, tem sua sintaxe modificada com frequência. Dessa forma, esse trabalho aborda a implementação de um analisador sintático escalável com o objetivo de tornar mais fácil e rápida modificações na sintaxe do formalismo de APEG.

## 1.1 Problema

APEG é um formalismo baseado em Parsing Expression Grammar (PEG) (FORD, 2004) que permite a modificação do conjunto de regras da gramática em tempo de análise do programa de entrada. APEG foi proposto recentemente e está em desenvolvimento, assim, a sintaxe concreta do formalismo ainda não foi concluída, passando por frequentes modificações.

No entanto, como formalismos para especificações de linguagens não são projetados de modo a facilitar modificações e manutenções, alterações no modelo é custoso. Portanto, o problema a ser abordado é a especificação da sintaxe de APEG com o intuito de facilitar mudanças e futuros aprimoramentos no formalismo.

## 1.2 Objetivo

O objetivo proposto neste trabalho é implementar um analisador sintático escalável para o formalismo APEG que possa ser facilmente modificado, resolvendo o problema de dificuldade de manutenção, tornando menos custoso o processo de modificação.

## 1.3 Metodologia

A metodologia utilizada segue como base a metodologia do artigo de Ryu (RYU, 2015). Foi utilizada a ferramenta Rats! (GRIMM, 2006) para a especificação modular da sintaxe e utilizado classes escritas à mão para a geração da AST. O objetivo era utilizar uma ferramenta para a geração automática da AST, mas houve alguns problemas, como a ferramenta ficar indisponível para download e falta de documentação, que levou a utilizar classes de construção da AST já desenvolvidas pelo autor do formalismo para a realização deste.

Primeiramente realizou-se um estudo da gramática de APEG para encontrar a melhor forma de modularizar a gramática. Nos capítulos posteriores é apresentada a razão dessa divisão. Em seguida, a especificação da sintaxe de APEG foi desenvolvida utilizando a ferramenta Rats!. Após concluída a especificação da sintaxe de APEG, foram elaborados um conjunto de casos de testes a fim de encontrar possíveis erros na especificação.

Concluída a fase de testes, realizou-se a correção dos erros encontrados obtendo-se a especificação final corrigida. Em seguida foi feita a especificação da AST, um novo conjunto de casos de testes foi elaborado e novamente o ajuste da sintaxe de acordo com os erros encontrados.

## 1.4 Contribuições

Os resultados alcançados neste trabalho são a especificação e modularização sistemática da gramática de APEG, observando as categorias sintáticas de cada regra para a criação de módulos apropriados que facilitem modificações porteriormente. Para desenvolver o analisador sintático utilizamos a ferramenta Rats! que possibilita a organização da gramática em módulos sintáticos. Desenvolvemos as classes da AST a partir do trabalho já iniciado pelo criador de APEG e por fim criamos testes para validar a especificação utilizando o JUnit para automatizar os testes.

## 1.5 Organização da Monografia

O texto será estruturado da seguinte maneira. No capítulo 2 será apresentado o formalismo de PEG e o formalismo de APEG. No capítulo 3, serão apresentados as ferramentas utilizadas para a realização deste trabalho, como foi feita a especificação escalável de APEG e como foi feita a especificação da AST. Por fim, como foram aplicados os testes sobre as especificações. Finalmente, o capítulo 4 apresenta a conclusão e trabalhos futuros.

# 2 Adaptable Parsing Expression Grammar

Neste capitulo, será introduzido o conceito do formalismo APEG. Para a melhor compreensão primeiramente vamos introduzir o conceito de PEG, tendo em vista que APEG é um modelo baseado em PEG. Encerraremos o capítulo com as conclusões.

## 2.1 Parsing Expression Grammar

PEG foi desenvolvida por Ford (FORD, 2004) tendo como finalidade descrever linguagens formais utilizando um conjunto de regras. Essas regras são usadas para reconhecer palavras dessa linguagem.

PEG é utilizado para expressar a sintaxe de linguagens de programação. PEGs se assemelham estruturalmente com gramáticas livres de contexto, possuindo alguns recursos de expressões regulares e notação Extended Backus-Naur Form. Possuindo interpretação diferente a de gramáticas livres de contexto, estando ligada diretamente a uma análise descendente recursiva. A principal diferença entre PEG e gramáticas livres de contexto é o recurso de escolha "prioritária", que diferente do operador "ou", o qual leva em consideração a ordem na qual os padrões são definidos. Outra diferença entre PEG e gramáticas livres de contexto é que PEGs não são ambíguas, isso significa que uma palavra que é aceita pela linguagem só possui uma árvore de derivação (parse tree) válida. Com isso PEG se torna muito útil para descrever linguagens de programação.

Formalmente, uma PEG pode ser definida como uma tupla ($\Sigma$, $N$, $P$, $S$) onde:

- $\Sigma$ é um conjunto finito de terminais
- $N$ é um conjunto finito de não-terminais
- $P$ é um conjunto finito de regras
- $S$ é uma expressão de análise chamada expressão inicial.

Cada regra $P$ segue a forma "$A \leftarrow e$" onde $A \in$ N é um símbolo não-terminal e $e$ é uma parsing expression. Uma parsing expression é uma expressão hierárquica similar a uma expressão regular, que é construída conforme o modelo descrito a seguir.

1. Uma parsing expression básica consiste de:
    - um símbolo terminal;
    - um símbolo não-terminal; ou

- a palavra vazia $\lambda$.

2. Dado que existem uma ou mais parsing expressions $e, e1$, e $e2$, uma nova parsing expression pode ser construida a partir dos seguintes operadores:

   - Sequência: $e1\ e2$
   - Escolha prioritária: $e1/e2$
   - Repetição: $e^*$
   - Predicado de negação: $!e$

Na Figura 1 apresentamos um exemplo de como utiliziar PEG para definir uma linguagem. Escolhemos apresentar como construir expressões matemáticas básicas (somente as quatro operações básicas), definindo o seu conjunto de regras.

```
1   Expressao   ← Soma
2   Soma        ← Produto ((‘+’ / ‘-’) Produto)*
3   Produto     ← Valor ((‘*’ / ‘/’) Valor)*
4   Valor       ← [0-9]+ / ‘(’ Expressao ‘)’
```

Figura 1 – Linguagem escrita em PEG

No exemplo acima pode-se observar que as regras são compostas de símbolos terminais e não-terminais. Os terminais são dados por caracteres “simples”, tal como ‘+’ e ‘-’, já os símbolos não-terminais são aqueles que se expandem para outras regras. São eles Expressão, Soma, Produto e Valor. Utilizamos de abreviações, que podem ser traduzidos para PEG com facilidade, a expressão [0-9] é um desses casos, indicando qualquer número no intervalo de 0 até 9. Essa sintaxe de abreviação é a mesma usada em expressões regulares.

Utilizaremos a expressão 1+2*3 para exemplificar como é feita a verificação de uma entrada para a linguagem especificada na Figura 1. Segue abaixo como ficaria o processamento da entrada.

```
1   Expressao   ← Soma;
2   Soma        ← Produto + Produto
3   Produto     ← Valor + Produto
4   Valor       ← 1 + Produto
5   Produto     ← 1 + Valor * Valor
6   Valor       ← 1 + 2 * Valor
7   Valor       ← 1 + 2 * 3
```

Figura 2 – Exemplo de processamento de uma entrada em PEG

A expressão tem como símbolo inicial "Expressao"que é substituido pela expressão do seu lado direito "Soma". "Soma"é composto por "Produto (( + / - ) Produto)*", dessa forma é feita a substituição por "Produto + Produto". A partir deste ponto a substituição é feita considerando o primeiro símbolo que aparece na regra. Na expressão "Produto + Produto"é substituido o primeiro "Produto"presente na expressão, cujo resultado é "valor + Produto", seguindo é substituido o símbolo "Valor", a expressão resultante é "1 + Produto". É importante notar que a substituição é feita sempre no primeiro símbolo que aparece na expressão até que o resultado seja o expressão desejada.

## 2.2 Adaptable Parsing Expression Grammar

O termo linguagens extensíveis é usado para referir a linguagens que possuem contruções que possibilitam extender sua propria sintaxe.

APEG é um formalismo que extende o modelo de PEG, tendo como objetivo descrever gramáticas extensíveis cujo conjunto de regras pode ser modificado em tempo de análise da palavra de entrada. APEG adiciona operadores ao conjunto original de PEG para permitir a extensão do conjunto de regras da gramática em tempo de análise do programa de entrada, facilitando a definição da sintáxe de linguagens extensívis (REIS; IORIO; BIGONHA, 2014).

O modelo possui uma notação chamada de *Atributo sintático*, sendo usados durante o processo de análise e seus valores são passados através de símbolos não-terminais da gramática. A gramática é um atributo do tipo valor de primeira ordem (*firt-order value*) passado para todo não-terminal, que no modelo é chamado de atributo de linguagem, e as regras usadas durante a análise são originadas deste atributo. A chave para extender dinamicamente a gramática da linguagem original é a habilidade de mudar e passar diferentes valores para o atributo de linguagem durante o processo de análise.

Como dito anteriormente, APEG possui *atribuitos sintáticos*, com isso o modelo faz a distinção entre dois tipos de atributos, sendo eles os herdados e sintetizados. Os atributos herdados tem seus valores definidos fora da regra não-terminal, agindo como um parâmetro de função. Por outro lado, os valores dos atributos sintetizados são definidos pela análise de um não-terminal no lado direito de uma regra. Dessa forma o atributo sintetizado é semelhante ao valor de retorno de funções.

A Figura 3 descreve a gramática da linguagem extensível *uSugar*. Um programa nessa linguagem começa com a possibilidade de declaração de novas regras, dado pela variável *newSyn*, seguido pela definição do nome e conteúdo da nova regra. Um comando pode ser atribuição, comando para leitura e escrita de valores, regras condicionais, de repetição ou comando de extensão da linguagem.

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

Figura 3 – Sintaxe da linguagem extensível *uSugar*

Como exemplo, na Figura 4 temos um programa escrito em *uSugar*. No começo do programa, linha 1 a 5, temos a definição de novas regras com nome *sfor*. Na segunda regra, linhas 3 e 4 é definida a sintaxe para a declaração da regra criada. Na linha 2 temos a definição que a nova regra pode ser usada como uma declaração de *uSugar* pelo comando *stmt* e na linha 3 temos a declaração da sintaxe da nova regra. O bloco da linha 7 a 11 é um exemplo da contrução de um laço de repetição (*for*) em *uSugar* a partir da declaração dada na linha feita na linha 3. No bloco da linha 13 a 17 é mostrado como seria o processamento da regra quando esta fosse executada, dando suas atribuições de valores e ordem de processamento.

```
1          define sfor {
2              stmt -> nfor;
3              nfor -> 'for' '(' attr ';' expr ';' attr ')'
4                  '{' stmt+ '}';
5          }
6
7          syntax sfor{
8              for (i := 1; i < 10; i := i + 1) {
9                  print i;
10             }
11         }
12
13 %        i := 1;
14 %        loop (i < 10) {
15 %           print i;
16 %            i := i + 1;
17 %        }
```

Figura 4 – Exemplo de programa em *uSugar*

## 2.3 Conclusão

Neste capítulo, foi apresentado as definições de PEG e APEG utilizando exemplos de como aplicar cada formalismo na construção de uma linguagem. Para entender,PEG se assemelha a gramáticas livres de contexto em estrutura tendo como diferença o processamento da entrada, onde em PEG é feita da esquerda para a direita, também chamado de ordem prioritátia, processando todo não-terminal a esquerda até consumir toda a entrada. APEG acrescenta operadores a PEG a fim de extender a gramática original em tempo de análise da entrada, podendo ser usadas para processar o restante dos símbolos de entrada.

# 3 Ferramentas Para Espeficicação Sintática

Neste capítulo será apresentado as ferramentas Rats! e o motivo pelo qual ela foi utilizada neste trabalho. Abordaremos como foi feita a especificação da sintaxe de APEG utilizando a ferramenta Rats! e a especificação da AST de APEG utilizando classes de criação da AST já desenvolvidas pelo autor no projeto do formalismo. Por fim, apresentaremos os testes aplicados sobre a especificação e os resultados obtidos.

## 3.1 Rats!

Rats! é um gerador de analisador sintático cuja gramática de entrada é baseada em PEG. Em se tratando de geradores de analisadores sintáticos, Rats! possui benefícios adicionais, entre eles, o mais expressivo para sua escolha nesse trabalho é que Rats! organiza gramáticas em módulos, ou seja, pode-se dividir a gramática em blocos, distintos ou relacionados, às regras da gramática, dessa forma caso ocorra uma alteração em determinada regra, sua modificação irá afetar somente o módulo ao qual ela pertence e aos módulos a ela relacionados, se existir. A alteração da regra será feita rapidamente, pois sabe-se a qual módulo a regra pertence, com isso ganha-se tempo já que não é necessário procurar onde está a regra no código-fonte. Dada essa vantagem, a escolha dessa ferramenta foi apropriada, já que propicia alcançar o objetivo proposto neste trabalho.

Nas Figuras 5, 6, 7 e 8 apresentamos um exemplo de como utilizar a sintaxe de Rats! para definir uma linguagem, para melhor contextualizar usaremos a mesma linguagem utilizada na Figura 1 presente no Capítulo 2.

Como pode ser observado no exemplo, Rats! permite separar a gramática em módulos. Foram criados quatro módulos distintos em Rats!, a criação do módulo se dá utilizando a palavra “module” seguido pelo seu nome completo. Note que existe dependências entre os módulos, por exemplo o módulo “expressao” é dependente do módulo “soma”, essa relação de dependência é dada em Rats! pelo uso da palavra “import” seguida pelo nome do módulo que se depende. A divisão em módulos é vantajosa, pois, caso ocorra uma mudança na sintaxe da gramática, essa vai afetar somente o módulo ao qual a mudança ocorre e aos seus dependentes. Em gramáticas com um grande número de regras, como é o caso de APEG, o tempo ganho na adequação da mudança será relevante, visto que sabendo a qual módulo a regra pertence e quais são seus dependentes não haverá necessidade de procurar em toda a gramática para fazer a modificação.

Tendo em mente que a gramática concreta de APEG ainda não foi concluída e passa por constantes modificações, o propósito de modularizar a gramática é tornar mais

```
1    module expressao;
2    import soma;
3
4    public void expressao = soma;
```

Figura 5 – Módulo Expressão

```
1    module soma;
2    import produto;
3
4    public void soma = produto (( '+' / '-' ) produto)*;
```

Figura 6 – Módulo Soma

```
1    module produto;
2    import valor;
3
4    public void produto = valor (( '*' / '/' ) valor)*;
```

Figura 7 – Módulo Produto

```
1    module valor;
2    import expressao;
3
4    public void valor = [0-9]+ / '(' expressao ')';
```

Figura 8 – Módulo Valor

fácil adaptações, novas construções ou a retirada de regras, visto que regras da gramática podem estar relacionadas entre si. Por exemplo, regras de espaçamento estão diretamente relacionadas a todas as outras regras da gramática. Dessa forma a criação dos módulos sintáticos foi feita observando as categorias sintáticas das regras. Por exemplo, regras relacionadas a expressões tem a mesma categoria sintática, tais como expressões aritméticas formadas por literais e operadores A gramática foi separada em 11 módulos sintáticos, a seguir estão apresentados os módulos e uma breve explicação de qual tipo cada um faz referência.

1. Módulo types: neste módulo estão as regras referente aos tipos existentes na linguagem, exemplo "int", "float", entre outros.

2. Módulo spacing: neste módulo estão as regras referentes à espaçamentos, quebra de linha, comentários em bloco e comentários de linha.

3. Módulo expression: neste módulo estão as regras referentes as expressões em geral, exemplo expressõs aritméticas, entre outras.

4. Módulo apeg_expression: neste módulo estão as regras referentes as parsing expressions de APEG.

5. Módulo attributes: neste módulo estão as regras referentes aos atributos de APEG.

6. Módulo functions: este módulo possui a regra referente a declaração de função.

7. Módulo apeg_definition: neste módulo estão presentes regras referentes as regras da gramática.

8. Módulo lexical: neste módulo estão as regras referentes aos literais da gramática, exemplo caracteres especiais, string literais.

9. Módulo metaprogramming: módulo referente aos operadores de manipulação de regras da gramática em tempo de análise da entrada.

10. Módulo option: neste módulo estão as regras sobre declarações opcionais, regras para declarar a definição da gramática, por exemplo cabeçalho.

11. Módulo grammar_rule: este módulo contém as definições das regras de APEG.

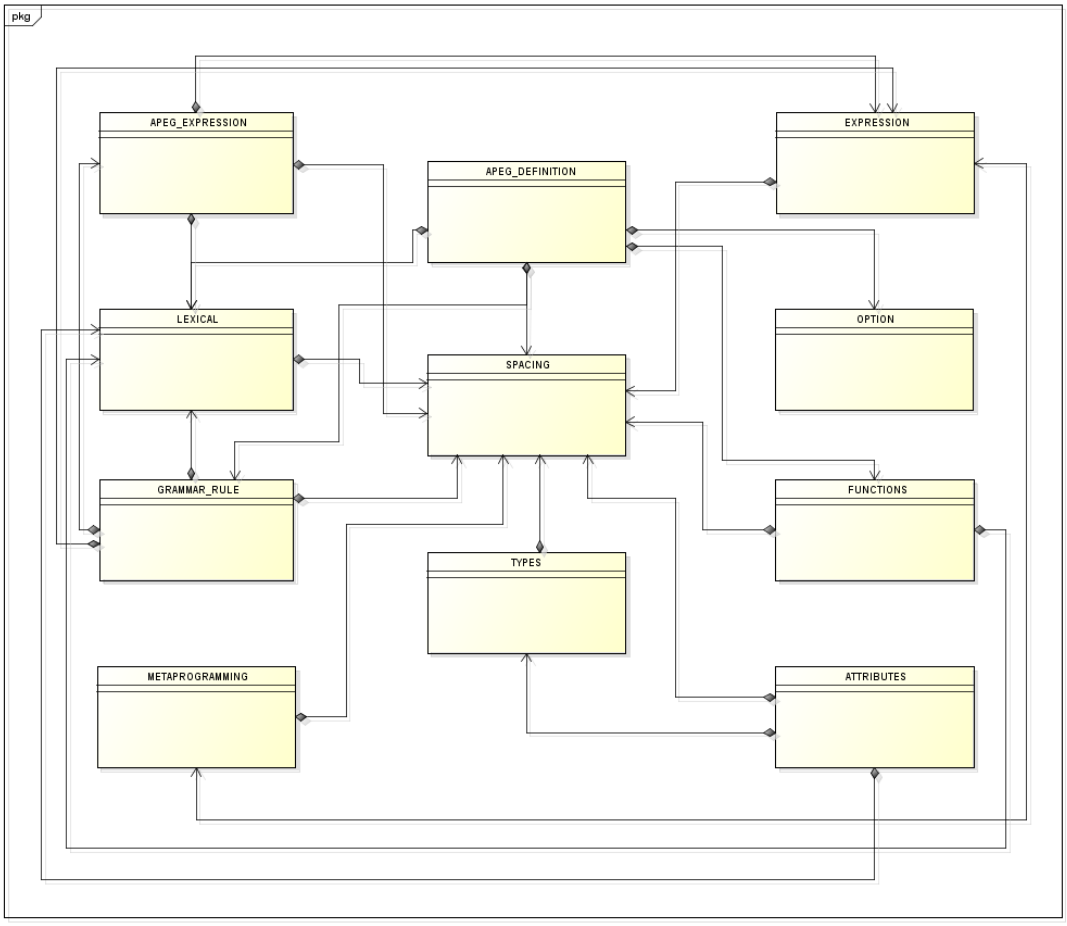


Figura 9 – Diagrama de pré-requisitos dos módulos da gramática de APEG

A Figura 9 utilizamos a notação de diagrama de classe para exemplificar o relacionamento de dependência dos módulos sintáticos da gramática. As setas indicam que o módulo de origem é dependente do módulo de chegada. Esse relacionamento acontece pelo fato de um módulo depender de regras de um ou mais módulos. Para demonstrar essa dependêcia vamos mostrar regras de dois módulos diferentes, abaixo a regra de espaçamento do módulo spacing:

```
1     public void ws = (‘ ’ / ‘\t’ / ‘\r’? ‘\n’ / ‘\r’)+;
```

A regra de espaçamento é representada com o nome “ws” fazendo referência a espaço, quebra de linha e tabulação. A seguir temos a regra referente a expressões de APEG do módulo apeg_expression:

```
1     public PegNode peg_expr =
2             e=peg_seq ws ‘/’ ws e1=peg_expr {yyValue = factory.
                 newChoicePeg(e, e1);}
3         |
4             e=peg_seq {yyValue = e}
5         ;
```

Como pode ser observado, a regra é nomeada “peg_expr” e em alguns pontos utiliza o não-terminal “ws” referente a regra “ws” do módulo spacing na sua composição. Esse detalhe caracteriza a dependência entre os módulos, onde o módulo apeg_expression depende do módulo spacing. Os módulos desenvolvidos podem ser encontrados no GitHub juntamente com o restante do trabalho.

## 3.2 ASTGen

Não foi possível utilizar a ferramenta ASTGen. Primeiramente iriamos utilizar a mesma ferramenta utilizada por Ryu em seu trabalho, mas a ferramenta ficou indisponível para download. Com isso procuramos outra ferramenta semelhante para substituirmos, encontramos um ASTGen desenvolvido por Panya (PANYAM, 2014), estudando o seu funcinamento não foi possível entender como utilizar a ferramenta, o autor não deixa claro como gerar automaticamente a AST e não disponibiliza uma documentação mínima para estudo do funcionamento.

Com esse contratempo, decidimos utiliziar classes de construição da AST já desenvolvidas pelo autor no projeto da gramática de APEG. Essas classes foram escritas em código Java e expecificam a criação da AST. Utilizamos o Rats! para criar automaticamente classes para serem testadas de forma similar a primeira expecificação da gramática.

## 3.3 Especificação de Testes

Foi escolhido aplicar testes com o objetivo de validar a especificação feita na gramática de APEG, dessa forma erros de especificação encontrados foram corrigidos. Para automatizar os testes foi utilizado o JUnit pois o Rats! gera código-fonte Java contendo as regras da gramática em forma de funções.

Um ponto importante e difícil dos testes foi a elaboração de casos de teste suficientes para validar a especificação, porque criar todos os casos de teste possíveis seria uma tarefa impossível. Foram elaborados um número de 207 casos de testes contendo palavras que deveriam ou não ser aceitas pela linguagem. Dessa forma já tinhamos o conhecimento prévio de quantos testes deveriam ser aceitos e rejeitados, com isso caso um teste não retornasse o valor esperado teriamos um caso de erro na especificação.

## 3.4 Conclusão

Neste capítulo mostramos como fizemos a construção dos módulos sintáticos utilizando para isso a ferramenta Rats!, que tem como característica organizar fragmentos da gramáticas em módulos. O critério utilizado para a divisão foi analisar as categorias sintáticas das regras, para buscar o melhor resultado para o trabalho. Por fim, apresentamos um exemplo de como uma gramática pode ser descrita utilizando a sua sintaxe.

# 4 Conclusão e Trabalhos Futuros

A principal motivação para a realização deste trabalho foi a necessidade de agilizar o processo de adaptação da gramática de APEG em caso de mudanças, dado que se trata de um formalismo recém desenvolvido e vem passando por constantes aprimorações. Para resolver esse problema, foi proposto a implementação do analisador sintático escalável de APEG. Os principais objetivos propostos foram:

1. dividir a gramática em módulos, facilitando modificações;
2. reduzir o tempo gasto com possíveis modificações;
3. gerar AST automaticamente, visto que é uma importante fase do projeto de um compilador e facilitar mudanças na estrututa da AST.

No início do trabalho a gramática de APEG era composta de um único módulo sintático e ao final realizamos a especificação e divisão da gramática em 11 módulos, observando suas categorias sintáticas. A especificação e modularização permite a adaptação e modificação de regras da gramática com um gasto reduzido de tempo caso seja necessário, não interferindo em regras que não estejam em módulos dependentes impossibilitando que a correção de uma regra cause erro em outra regra.

O teste automatizado foi um recurso que forneceu a validação necessária de que a especificação foi feita corretamente e não possui erros, de forma que a AST gerada está correta, podendo assim passar para as próximas fases de criação do compilador.

A geração automática da AST seria realizada com o auxílio da ferramenta ASTGen, utilizariamos a mesma que Ryu utilizou em seu trabalho, porém a ferramenta ficou indisponível para download. Dessa forma a segunda opção foi procurar uma outra ferramenta semelhante para substituirmos, encontramos a ferramenta ASTGen (PANYAM, 2014), mas essa não continha uma documentação para ser estudada, tornando inviável a sua utilização. Como ficamos sem recursos de ferramentas, a solução foi utilizar classes criadas manualmente para concluir o trabalho, o que de certa forma prejudicou um pouco os resultados que queriamos obter, já que sem a geração automática da AST não foi possível fazer uma validação adequada do resultado obtido com este trabalho.

## 4.1 Trabalhos futuros

Como foi dito anteriormente, a geração da AST é uma das fases do projeto de um compilador. Visto que APEG é um formalismo novo, a implementação de um compilador

que consiga interpretar sua gramática é o seguinte trabalho a ser feito.

Uma validação pode ser feita posteriormente da seguinte forma: modificarir uma regra da gramática, fazer a adaptação manualmente e utilizando uma ferramenta de geração automática da AST, computando o tempo gasto, o número de regras que seriam modificadas no processo. Com esses números fariamos uma comparação para conhecer o tempo gasto ganho e se o total de regras que sofreriam adaptação seria menor que feito manualmente.

# Referências

FORD, B. Parsing expression grammars: a recognition-based syntactic foundation. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2004. v. 39, n. 1, p. 111–122. Citado 2 vezes nas páginas 21 e 23.

GRIMM, R. Better extensibility through modular syntax. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 2006. v. 41, n. 6, p. 38–51. Citado na página 22.

PANYAM, S. *ASTGen*. 2014. Disponível em: <https://github.com/panyam/astgen/>. Último acesso em Março de 2017. Citado 2 vezes nas páginas 32 e 35.

REIS, L. S. et al. Adaptable parsing expression grammars. *Programming Languages, in: Lecture Notes in Computer Science*, v. 7554, p. 72–86, 2012. Citado na página 21.

REIS, L. V.; IORIO, V. O. D.; BIGONHA, R. S. Defining the syntax of extensible languages. In: ACM. *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. [S.l.], 2014. p. 1570–1576. Citado na página 25.

RYU, S. Scalable framework for parsing: from fortress to javascript. *Software: Practice and Experience*, Wiley Online Library, 2015. Citado 4 vezes nas páginas 11, 13, 21 e 22.