UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Henrique Grolli Bassotto

EXTENSÃO DA LINGUAGEM E IMPLEMENTAÇÃO DE INTERPRETADOR PARA ORDERLY

Florianópolis - SC

2013

Henrique Grolli Bassotto

EXTENSÃO DA LINGUAGEM E IMPLEMENTAÇÃO DE INTERPRETADOR PARA ORDERLY

TCC submetido ao Curso de Bacharelado em Sistemas de Informação para a obtenção do Grau de Bacharel.
Orientador: Prof. Dr. Olinto José Varela Furtado

Florianópolis - SC

2013

Henrique Grolli Bassotto

EXTENSÃO DA LINGUAGEM E IMPLEMENTAÇÃO DE INTERPRETADOR PARA ORDERLY

Este TCC foi julgado aprovado para a obtenção do Título de "Bacharel", e aprovado em sua forma final pelo Curso de Bacharelado em Sistemas de Informação.

Florianópolis - SC, 18 de fevereiro 2013.

Renato Cislaghi
Coordenador

Prof. Dr. Olinto José Varela Furtado
Orientador

Banca Examinadora:

Profa. Dra. Jerusa Marchi
Presidente

Prof. José Eduardo De Lucca

SUMÁRIO

1 INTRODUÇÃO 7
1.1 OBJETIVOS 8
1.2 ESTRUTURA DO TRABALHO 8
2 FUNDAMENTAÇÃO TEÓRICA 11
2.1 COMPILADORES 11
2.1.1 Análise Léxica 12
2.1.2 Análise Sintática 12
2.1.3 Análise Semântica 13
2.2 GERADOR DE PARSER 13
2.2.1 Gramáticas no ANTLR 14
2.3 JSON 16
2.3.1 Sintaxe JSON 16
2.3.1.1 Objetos 16
2.3.1.2 Strings 16
2.3.1.3 Arrays 17
2.3.1.4 Números 17
2.3.1.5 Outros Valores 18
2.3.2 JSON Schema 18
2.4 ORDERLY 18
2.4.1 A Sintaxe da Linguagem Orderly 20
2.4.1.1 Propriedades Comuns 20
2.4.1.2 Strings e Números 21
2.4.1.3 Objetos 22
2.4.1.4 Arrays 22
2.4.1.5 Demais Tipos 23
3 EXTENSÃO DE ORDERLY 25
3.1 DEFICIÊNCIAS DA LINGUAGEM ORDERLY 25
3.1.1 Necessidade de Compilação 25
3.1.2 Suporte a Referências 25
3.1.3 Dependência entre Propriedades 25
3.2 ALTERAÇÕES NA GRAMÁTICA 26
3.2.1 Implementação de Referência (Import) 26
3.2.2 Implementação de Dependência (Requires) 27
3.2.3 Alterações em Arrays 28
3.2.4 Propriedades Opcionais 30
4 INTERPRETADOR ORDERLY 31
4.1 PARSER ORDERLY 32

4.1.1 Árvore Intermediária 33
4.1.1.1 JSONObject e Verificação de Referência 34
4.2 PARSER JSON 35
4.3 TESTES E DEBUGGING 37
4.3.1 Debug Avançado da Gramática 38
5 USANDO O INTERPRETADOR 41
6 CONCLUSÕES E TRABALHOS FUTUROS 43
6.1 DEPENDÊNCIA ATRAVÉS DE EXPRESSÕES 44
6.2 DECLARAÇÃO DE TIPOS E REFERÊNCIA INTERNA .. 44
6.3 AÇÕES SEMÂNTICAS EM ORDERLY 45
6.4 REPRESENTAÇÃO DISCRETA DO DOCUMENTO JSON 46
6.5 INTERPRETADOR PARA OUTRAS LINGUAGENS 46
REFERÊNCIAS 47
APÊNDICE A – Gramática do JSON 51
APÊNDICE B – Gramática Orderly 55
APÊNDICE C – Script de Testes 61

1 INTRODUÇÃO

Serialização de dados é necessária em quase todas as aplicações de um software. Tanto em sistemas Web quanto em programas locais, é necessário representar estruturas de dados que serão armazenadas ou transmitidas. Várias formas de serialização existem sendo, na Web, as mais populares XML e JSON (MALESHKOVA; PEDRINACI; DOMINGUE, 2010).

Apesar de popular e estabelecida a notação XML vem perdendo espaço para uma alternativa criada por volta de 2001. O formato *Javascript Object Notation* ou *JSON* foi proposto por Douglas Crockford como um sub conjunto da linguagem JavaScript e encontra-se altamente difundido na web sendo preferido para implementação de APIs públicas junto com XML (MALESHKOVA; PEDRINACI; DOMINGUE, 2010).

Motivos para adoção do JSON podem ser atribuídos à sua praticidade e simplicidade. Comparado a XML, JSON apresenta uma estruturas voltada ao encapsulamento dos dados mais limpa e é, discutivelmente, mais legível que XML para a maioria dos casos. Outra grande vantagem é a facilidade com que este pode ser interpretado tendo virtualmente em todas as linguagens uma estrutura de dados nativa compatível.

Independente da forma com que se serialize os dados, uma grande dificuldade e fonte de falhas em APIs é a validação destes dados e a documentação desses formatos (MALESHKOVA; PEDRINACI; DOMINGUE, 2010). É extremente trabalhoso implementar manualmente a validação dos dados para estruturas complexas. Por consequência, a manutenção desse código é igualmente complexa. Nesse contexto reside uma das grandes diferenças entre XML e JSON pois a primeira possui em sua especificação o DTD (Document Type Definition) que formaliza quais elementos podem existir e de que forma estes se organizam no documento. Há ainda uma outra especificação oficial para o mesmo fim que permite a adição de restrições mais complexas para os elementos. Esta especificação se chama XML Schema e nada mais é que um documento XML onde estão descritos os elementos e o formato dos seus valores.

Para adicionar essa mesma capacidade de validação presente em XML foi proposta a especificação JSON Schema (ZYP; COURT, 2010) que, diferente da especificação XML Schema na qual foi baseada, não obteve grande adoção e encontra-se em estado de abandono. Para corrigir as deficiências do formato JSON Schema a linguagem Orderly foi criada por Lloyd Hilaiel (HILAIEL, 2012) para permitir a elaboração de

documentos JSON Schema de forma mais legível e expressiva através de um processo de compilação que transforma documentos Orderly em documentos JSON Schema. Contudo, ao manter a dependência com JSON Schema através da compilação a linguagem Orderly perde a possibilidade de implementar recursos novos. Outro problema encontrado é a falta de praticidade de se usar algo compilado em linguagens de programção interpretadas como Python, Ruby ou Javascript.

Este trabalho ataca as principais deficiências de Orderly ao desenvolver e implementar um interpretador para a linguagem permitindo o uso de documentos Orderly de forma direta, sem a necessidade de compilação. Removendo também a dependência com JSON Schema novos recursos para a linguagem são propostos e implementados.

1.1 OBJETIVOS

O objetivo principal deste trabalho é desenvolver extensões e adaptações à linguagem Orderly a fim de resolver deficiências comentadas na introdução, e detalhadas na Seção 3.1, melhorando a expressividade e praticidade da linguagem.

Para facilitar o uso do Orderly é objetivo deste trabalho também o desenvolvimento de um interpretador que elimine a necessidade de compilação para JSON Schema, desacoplando a linguagem e permitindo a sua evolução sem amarras de compatibilidade.

Os seguinte objetivos específicos serão trabalhados:

1. Estender a gramática da linguagem Orderly para suportar referência externa.

2. Estender a gramática da linguagem Orderly para suportar dependência condicional.

3. Implementar de um interpretador Orderly com as extensões planejadas para validação de dados JSON.

4. Testar o uso da linguagem Orderly e do interpretador desenvolvido em um ambiente real.

1.2 ESTRUTURA DO TRABALHO

Para descrever a implementação do interpretador apresentada no Capítulo 4 é primeiramente apresentada no Capítulo 2 a teoria

básica sobre compiladores e as tecnologias envolvidas nesse trabalho como ANTLR, JSON, e a linguagem Orderly. Após isso as extensões da linguagem que foram propostas e implementadas são detalhadas no Capítulo 3. Após os detalhes da implementação mostrados no Capítulo 4 é apresentado o resultado do interpretador do ponto de vista do usuário no Capítulo 5 e por fim são levantadas as conclusões e sugestões para trabalhos futuros no Capítulo 6.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 COMPILADORES

De acordo com o livro "Compiladores: Princípios, técnicas e ferramentas"(AHO et al., 2010), compiladores são softwares que fazem a tradução de uma linguagem de programação para um formato compreensível por um computador. Para tanto são usados conceitos de linguagens de programação, teoria de linguagens formais, algoritmos e engenharia de software. O compilador faz isso recebendo como entrada um programa escrito em uma linguagem de programação — programa fonte — e o traduz para outro programa equivalente em outra linguagem — programa objeto.

Se o programa objeto gerado pelo compilador for um programa em linguagem de máquina executável este programa poderá ser chamado pelo usuário para processar uma entrada e produzir uma saída. Um interpretador por sua vez é outro tipo comum de processador de linguagem — como um compilador — que ao invés de traduzir um programa fonte para um programa objeto executa diretamente o programa fonte e uma entrada gerando a saída desejada.

Interpretadores tendem a ser muito mais lentos que compiladores uma vez que precisam processar o código fonte em cada execução. Para reduzir essa lentidão pode-se combinar as técnicas de geração de código de máquina e interpretação. Essa técnica é usada por diversos interpretadores populares como a máquina virtual java, python e várias implementações javascript, como a V8 usada no browser chrome.

Um compilador/interpretador pode ser descrito como um conjunto de fases onde cada uma faz uma transformação em uma entrada que por sua vez serve de entrada para a próxima fase. Essas fases são agrupadas em duas etapas: análise e síntese. A etapa de análise é composta das fases de análise léxica, sintática e semântica e é onde o programa fonte é lido e validado gerando uma saída adequada para as próximas fases. A síntese é a parte do compilador que faz a geração do programa objeto e não é relevante a compreendermos em detalhes nesse trabalho uma vez que o interpretador desenvolvido não faz geração de código.

A implementação da análise é geralmente feita de forma agrupada onde a análise léxica e semântica são realizadas na medida em que a análise sintática progride. Essa técnica é conhecida como tradução dirigida por sintaxe (AHO et al., 2010).

2.1.1 Análise Léxica

A primeira fase de análise em um compilador é chamada *análise léxica*. Essa etapa recebe o fluxo de caracteres do programa fonte e os agrupa em sequências significativas chamadas *lexemas*. Esses lexemas são representados e passados para a próxima fase de compilação com o nome de *token*. Essa abstração é importante para facilitar a escrita e compreensão de uma gramática para suas especificações sintáticas e semânticas. Tomemos como exemplo a seguinte linha de código:

```
variable = base * 2;
```

Um analisador léxico transforma essa entrada em uma sequência de tokens conforme apresentado abaixo:

```
IDENTIFIER(variable) = IDENTIFIER(base) OPERATOR(*) NUMBER(2);
```

Essa transformação permite abstrair muito mais facilmente as regras de sintaxe do programa.

2.1.2 Análise Sintática

Análise sintática é a etapa de compilação que faz a análise do fluxo de tokens vindos da etapa de análise léxica e avalia se os mesmos estão na ordem correta. Essa ordem é definida por várias regras de sintaxe que vão determinar o que é permitido pelo programa fonte que se deseja analisar. Considerando o exemplo apresentado na seção anterior, tem-se abaixo uma representação comum para atribuição de variável:

```
expression : IDENTIFIER (OPERATOR expression)?
           | NUMBER (OPERATOR expression)?
           ;

command : IDENTIFIER = expression
        ;
```

As regras sintáticas determinam como os tokens — representados em caixa alta — devem ser organizados. A definição das regras sintáticas são a parte da construção de um compilador que tem maior influência sobre o formato da linguagem. Essa influência vem

do fato destas regras determinarem como são construídas todas as declarações presentes na linguagem como a definição de classes, métodos e operações de chamada de função. Também são determinados os formato dos elementos de controle de fluxo como *for*, *if* e *while*.

2.1.3 Análise Semântica

A análise semântica é a última fase de análise e tem um papel importante na construção de um compilador. Essa etapa é responsável por garantir que o significado básico da linguagem seja respeitado. Por exemplo, em uma atribuição de valor como mostrada no exemplo da Seção 2.1.1, é reponsabilidade do analisador léxico garantir que os caracteres corretos estejam sendo utilizados na entrada, agrupando-os em tokens. O analisador sintático então, garante que a estrutura da declaração esteja correta para que o analisador semântico verifique se o sentido da atribuição é válido. Isso é necessário para, por exemplo, verificar se somente variáveis já declaradas estejam sendo usadas ou que os tipos dos identificadores usados em uma expressão sejam compatíveis com o operador escolhido.

Para que seja possível fazer a análise semântica, o compilador tem um componente chamado *tabela de símbolos*. Este componente armazena informações sobre o código fonte que são usadas para as verificações semânticas. Nessa tabela são armazenados identificadores, tipos de variáveis, métodos e classes. No caso do interpretador Orderly, apresentado neste trabalho, a tabela de símbolos é representada pela árvore intermediária apresentada na Seção 4.1.1.

Apesar de as regras semânticas poderem ser descritas de forma declarativa como são feitas as regras de sintaxe não existem técnicas adequadas para problemas complexos, por isso elas devem ser implementadas de forma manual. Por serem programadas ao invés de declaradas elas são chamadas de *ações semânticas*. Esse é o motivo pelo qual as gramáticas deste trabalho apresentam código java em suas definições como pode ser observado na Seção 4.1.

2.2 GERADOR DE PARSER

Parsers não precisam ser implementados diretamente. É possível usar um software chamado *parser generator*, um tipo de compilador que transforma uma definição de regras léxicas, sintáticas e semânticas,

tipicamente em BNF[1], em um parser.

Diversos geradores de parser estão disponíveis de forma gratuita. Os geradores avaliados para esse trabalho foram o GALS, JAVACC, ANTLR e BISON.

GALS é um gerador de parser desenvolvido por Carlos Gesser (2003) e estendido por Ronaldo Campos (2009) (CAMPOS, 2009). Possui uma interface gráfica e pode gerar diversos tipos de analisadores. Sua forma de lidar com análise semântica é feita usando tradução dirigida pela sintaxe onde, ao encontrar um símbolo de análise semântica, o analisador interrompe a análise chamando a regra da ação passando como parâmetro o último token identificado.

JavaCC é um gerador de parser feito em Java e pode gerar analisadores com código em Java e C++.

Bison é um gerador de parser parte do projeto GNU (FSF, 2012). Bison lê o conteúdo de uma gramática livre de contexto convertendo a mesma para um analisador LALR em C, C++ ou Java.

ANTLR é um gerador de parser que gera analisadores LL. Uma das suas maiores vantagens é possuir suporte a muitas linguagens alvo incluindo Java, C/C++, Python e Javascript. Sua documentação é completa e seu desenvolvimento é ativo (PARR, 2012).

O ANTLR foi escolhido como gerador de parser deste trabalho por ter documentação mais completa trazendo maior segurança de manutenibilidade. Também foi levado em consideração a capacidade de gerar código não só em Java, que é uma restrição do ambiente descrito na Seção 4, mas em diversas outras linguagens permitindo assim uma melhor extensibilidade deste trabalho.

2.2.1 Gramáticas no ANTLR

As gramáticas usadas para descrever linguagens no ANTLR são construídas usando EBNF (Extended BNF). Como visto anteriormente ANTLR implementa um parser LL descendente recursivo, isso traz diversas restrições para a implementação das gramáticas.

Por gerar um parser descendente recursivo LL a seguinte gramática, extraída do livro "The Definite ANTLR Reference"(PARR, 2007), não funciona com ANTLR:

```
expr : expr '++'
     ;
```

[1] *Backus–Naur Form* e suas variações são populares formas de se descrever gramáticas livre de contexto.

A gramática acima não funciona pois, quando transformada em código, terá uma recursão à esquerda que fará um loop infinito como mostrado abaixo:

```
void expr() {
    expr();
    match("++" );
}
```

Gramáticas LL também trazem algumas limitações quanto ao formato de declarações ambíguas. A declaração abaixo por exemplo não poderia ser reconhecida em um parser LL puro por diversos motivos:

```
decl : 'int' declarator '=' INT ';' // Ex: "int **x=3;"
     | 'int' declarator ';' // Ex: "int *x;"
     ;

declarator // Ex: "x", "*x", "**x", "***x"
     : ID
     | '*' declarator ;
```

Primeiro temos uma ambiguidade na declaração de *decl* onde a primeira parte da declaração não poderia ser reconhecida. Essa limitação é resolvida usando uma técnica chamada *lookahead* que faz uma leitura antecipada dos próximos tokens da entrada a fim de determinar posteriormente qual a alternativa de reconhecimento será usada. Enquanto técnicas normais de lookahead são feitas com limitações específicas da quantidade de símbolos analisados LL(k), ANTLR implementa um lookahead dinâmico chamado de LL(*) que faz a pesquisa de um número arbitrário de símbolos para resolver a ambiguidade (PARR, 2007).

Mesmo com técnicas de lookahead LL(*) a gramática acima tem um problema de recursão na declaração *declarator* que faz com que a implementação de *backtrack* seja necessária para evitar a fatoração manual das declarações. Backtrack é a técnica de voltar atrás ao detectar que está em uma alternativa errada e tentar a próxima candidata. Essa prática é extremamente cara em termos de complexidade algoritimica, podendo alcançar uma complexidade exponencial no pior caso. Para isso ANTLR implementa outra técnica onde o trabalho feito na primeira análise errada é mantido em memória para facilitar as próximas análises melhorando muito o desempenho do backtrack (PARR, 2007).

Com essas e outra técnicas implementadas pelo ANTLR as gramáticas feitas para esse gerador de parser podem ser feitas sem muitas res-

trições e refatoramentos facilitando muito a criação e compreensão das gramáticas.

2.3 JSON

JavaScript Object Notation (JSON) é um formato de serialização de dados projetado por Douglas Crockford (CROCKFORD, 2006). Apesar da estrutura do JSON ser baseada na sintaxe da declaração literal de objetos da linguagem JavaScript, não há restrições para que este formato de serialização seja usado por sistemas escritos em outras linguagens.

2.3.1 Sintaxe JSON

Como descrito em sua definição(CROCKFORD, 2008) JSON possui seis tipos de valores: Objetos, Arrays, Strings, Números, Booleanos e o valor especial *null*. Espaços em branco e novas linhas podem ser inseridos antes e depois de qualquer valor permitindo formatar o documento fonte de forma a melhorar sua legibilidade. Esses caracteres podem ser suprimidos para reduzir custos de armazenamento e transmissão de dados quando conveniente. A Figura 1 mostra um documento JSON com seus diversos tipos de dados.

2.3.1.1 Objetos

Um objeto JSON é uma lista não ordenada de pares chave/-valor. Uma chave deve ser um string e valor pode assumir qualquer valor JSON incluindo outros objetos que podem ser aninhados a qualquer profundidade. A maioria das linguagens de programação possui estruturas de dados que podem ser facilmente traduzidas para objetos JSON, como objetos nativos, arrays, dicionários, hash tables, arrays associativos e outros.

2.3.1.2 Strings

Um string JSON é uma sequência de caracteres encapsuladas por aspas duplas. A barra invertida "\" é usada como caractere de controle como mostra o *propriedade2* no exemplo da Figura 1.

```
{
  "chave": "valor",
  "propriedade2" : "string complexa com \" escapes"
  "objeto" : {
     "outro" : "valor",
     "numero" : 123.34,
     "avogrado" : 6.02e23
  },
  "array" : [1, 2, 3, 4, 5],
  "array complexo" : [
    12,
    13.3,
    "string",
    [3, 2],
    {"me": "object"}
  ],
  "nulo" : null,
  "falso" : false,
  "verdadeiro" : true
}
```

Figura 1: Exemplo de dados serializados em JSON

2.3.1.3 Arrays

Um array JSON é uma sequência ordenada de valores de qualquer tipo e pode ser mapeada na maioria das linguagens em forma de arrays, listas, vetores, etc.

2.3.1.4 Números

Numeros em JSON são como números em JavaScript, porém não é permitido adição de 0 no início de inteiros pois algumas linguagens usam isso para indicar um octal. Números podem ser inteiros, reais ou em notação científica como demonstra as propriedades *numero* e *avogrado* no exemplo apresentado na Figura 1.

2.3.1.5 Outros Valores

O tipo booleano é usado através de dois literais: true e false. Representando verdadeiro e falso respectivamente. O valor especial null é usado para indicar nulidade e possui elemento semelhante na maioria das linguagens.

2.3.2 JSON Schema

Em uma tradução livre da sua especificação (ZYP; COURT, 2010) JSON Schema é descrita como uma notação que define um novo *media type* "application/schema+json", um formato baseado em JSON para definição de estrutura de dados em JSON. JSON Schema provê um contrato ao qual aplicações podem referenciar ao lidar com dados serializados em JSON. JSON Schema tem a intenção de definir validação, documentação, navegação e controle de interação para dados em JSON.

O documento da Figura 2 adaptado de (ZYP; COURT, 2010) é um exemplo de uma especificação em JSON Schema que descreve um objeto *Product* com três propriedades obrigatórias (id, name e price) e uma propriedade opcional (tags).

Mesmo com sua interpretação sendo relativamente fácil, o formato JSON Schema não ganhou muitos adeptos, isso está evidenciado no abandono de sua proposta (ZYP; COURT, 2010) invalidada como internet draft na IETF[2] desde 2011 ao contrário de JSON que teve a sua RFC[3] publicada como padrão para a Internet em 2006.

2.4 ORDERLY

Orderly é uma linguagem criada por Lloyd Hilaiel (HILAIEL, 2012) desenhada para oferecer uma forma mais expressiva e legível de especificar formato de um documento JSON mantendo a compatibilidade com JSON Schema. O que ela faz basicamente é condensar as várias propriedades existentes no JSON Schema em estruturas sintáticas mais concisas.

[2]Internet Engineering Task Force é uma organização que discute e publica técnicas e boas práticas para a Internet. Seu objetivo é definido como: "Fazer a internet funcionar melhor".

[3]Request For Comments são memorandos publicados pela IETF descrevendo métodos, comportamentos, pesquisa ou inovação aplicável à Internet. Com apoio e revisão suficiente uma proposta é adotada como padrão para a Internet.

```
{
  "name":"Product",
  "type" : "object",
  "properties":{
    "id":{
      "type":"number",
      "description":"Product id",
      "required":true
    },
    "name":{
      "description":"Product name",
      "type":"string",
      "required":true
    },
    "price":{
      "required":true,
      "type": "number",
      "minimum":0,
      "maximum": 1000
    },
    "tags":{
      "type":"array",
      "items":{
        "type":"string"
      }
    }
  }
}
```

Figura 2: JSON Schema

A Figura 4 mostra um documento Orderly implementando as mesmas especificações do JSON Schema apresentado na Figura 2, ambos validam o documento JSON representado na Figura 3. O fato de JSON Schema ser um documento JSON obriga a notação a ser extremamente verbosa. Em Orderly algumas características, tais como restrições de limites para números, podem ser descritas usando poucos caracteres como mostra a propriedade *price*: enquanto em JSON Schema ela necessita de uma estrutura com várias propriedades, em Orderly essa definição tem uma única linha.

```
{
  "id" : "0553382578",
  "name" : "Foundation",
  "price" : 39.0,
  "tags" : [
    "Asimov",
    "Foundation",
    "Science Fiction"
  ]
}
```

Figura 3: JSON válido para o Schema da Figura 2

```
// Object Product
object {
  number id;
  string name;
  number {0,1000} price;
  array [string]* tags;
}*;
```

Figura 4: Especificação Orderly. Essa especificação é equivalente ao JSON Schema da Figura 2

2.4.1 A Sintaxe da Linguagem Orderly

A sintaxe da linguagem Orderly [4] é muito simples e tem sua estrutura base semelhante a do JSON. Ela permite a representação dos seis tipos de valores da notação JSON como apresentado na Seção 2.3.1 e mais dois tipos especiais: *any* e *union*.

2.4.1.1 Propriedades Comuns

Quatro propriedades se aplicam a todos os tipos de dados.

[4] Descrição de formato Orderly baseado em informações extraídas do site oficial (HILAIEL, 2012) em Junho de 2012. A fim de manter a consistência do trabalho uma cópia da gramática como se encontra na data está inclusa no Anexo B.

- Um ponto de interrogação pode ser usado opcionalmente após o identificador da propriedade para definir que ela é opcional, ou seja, que a mesma pode existir em um documento válido ou ser omitida da estrutura JSON final.

  ```
  string comentario?;
  ```

- Após a definição do identificador da propriedade pode se inserir o identificador de outra propriedade entre os caracteres <e >para indicar que a propriedade é obrigatória se o identificador referenciado existir. Essa relação de dependência será explicada melhor na Seção 3.1.3.

  ```
  string cargo<empresa>;
  ```

- Suporte a enumeração é feita passando uma lista de valores permitidos a uma propriedade separados por virgula e encapsuladas por colchetes. O conteúdo da lista deve respeitar o tipo declarado.

  ```
  string tipo["simples","composto","especial"];
  ```

- Um valor padrão pode ser definido para uma propriedade em uma estrutura parecida com a de atribuição em linguagens de programação.

  ```
  string name = "Não Definido";
  ```

2.4.1.2 Strings e Números

Strings são especificadas em Orderly pelo tipo *string*, números pelos tipos *number* e *integer*. Além das propriedades comuns apresentadas na Seção 2.4.1.1 strings e números possuem o modificador de intervalo. Para strings o intervalo representa o tamanho, em caracteres, que se deseja representar.

```
string{4,} login; # string login com minimo de 4 caracteres
string{,32} name; # string name com máximo de 32 caracteres
integer {0, 100} percentage; # número que varia de 0 à 100
number {0, 0.5} percentage; # número que varia de 0.0 à 0.5
```

Ainda pode-se especificar o conteúdo permitido em um string através de expressões regulares. Estas são representadas encapsuladas pelo caractere “/” para delimitar seu começo e fim.

```
string email /^[a-Z0-9._%+-]+@[a-Z0-9.-]+\.[a-Z]{2,4}$/;
```

2.4.1.3 Objetos

Objetos são definidos pelo tipo *object* onde as propriedades do objeto são definidas como uma lista separada pelo caracter “;” e encapsulado por chaves. Um asterisco ao final da chave que fecha a lista de propriedades significa que o objeto aceita propriedades adicionais sem restrições.

```
object {
    string foo;
    integer bar;
    number baz;
}*;
```

2.4.1.4 Arrays

Arrays são declarados com o tipo *array* e podem ser especificados de forma simples onde basta definir o tipo aceito pelo array. Também é suportado definição de intervalo, assim como em strings e números mas que determinam a quantidade de elementos no array.

```
# um array de pesos representados com numeros reais entre 0 e 1.
array [number{0.00, 1.00}] weights;

# Array de strings contendo no máximo 10 elementos.
array [string] {,10};
```

Ainda é possível determinar arrays que suportem mais de um tipo substituindo os colchetes por chaves.

```
array {
    number;
    string;
} many; # array de numeros e strings
```

2.4.1.5 Demais Tipos

Orderly apresenta dois tipos de valores especiais. São eles *any* e *union*. Ambos são usados para mapear uma composição de valores básicos. *Any* indica que qualquer tipo básico pode ser usado com a propriedade definida. Já o valor do tipo *union* indica tipos especificos que podem ser usados na propriedade.

```
// objeto que possui uma propriedade name que
// pode ser somente um string alfanumérico
// ou o valor null
object {
  union {
    string /^[a-zA-Z0-9]+$/;
    null
  } name;
  boolean alive;
}
```

3 EXTENSÃO DE ORDERLY

3.1 DEFICIÊNCIAS DA LINGUAGEM ORDERLY

3.1.1 Necessidade de Compilação

Orderly foi desenhada para ser compilável para JSON Schema ao invés de ser interpretada. Disso deriva sua maior fraqueza já que além de ficar limitada aos recursos oferecidos pelo JSON Schema, há a necessidade de uma etapa extra de compilação. Somente após essa estapa é possível fazer uso da especificação para validação de dados.

A complexidade adicional do uso de uma etapa de compilação poderia ser removida com o uso de um interpretador que faria a validação de dados JSON diretamente através da compreensão de uma entrada Orderly.

3.1.2 Suporte a Referências

Um importante dispositivo de reuso de código usado em linguagens de programação são as referências. Geralmente associadas a funções, elas tem a tarefa de reproduzir em mais de um ponto do código algo que já foi definido anteriormente sem a necessidade de reimplementação. Em Orderly não há atualmente suporte a referências apesar de as mesmas existirem em JSON Schema. O exemplo abaixo mostra uma forma em que referência poderia ser implementada.

```
import "Pessoa.orderly";
object {
  string nome;
  array [Pessoa] contatos;
} agenda;
```

3.1.3 Dependência entre Propriedades

Existe em Orderly, por herança do JSON Schema, uma definição de dependência simples onde uma propriedade depende da existência de outra. Porém não há forma de representar dependência de um valor específico entre duas propriedades. Por exemplo no seguinte documento

Orderly não há forma de definir uma dependência de obrigatoriedade para a propriedade *classificacao* somente se o valor da propriedade *categoria* for, por exemplo, “filmes”.

```
object {
    string nome;
    string categoria;
    string classificacao;
};
```

A declaração de relações de dependência condicional poderia ser implementada como mostra o exemplo abaixo.

```
object {
    string nome;
    string categoria;
    string classificacao <categoria = "filmes">;
};
```

3.2 ALTERAÇÕES NA GRAMÁTICA

Para resolver os problemas detectados na Seção 3.1 foi implementado um interpretador capaz de validar documentos JSON através de documentos Orderly.

Algumas alterações adicionais foram feitas na especificação da linguagem Orderly. Essas alterações tem o objetivo de facilitar a implementação das extensões propostas ao mesmo tempo melhorarando a expressividade e legibilidade da linguagem.

3.2.1 Implementação de Referência (Import)

Das extensões propostas, a possibilidade de importar outros documentos é a mais importante pois permite abstrair a complexidade de alguns protocolos em diferentes arquivos permitindo o reuso de definições.

Como foi implementada, a importação e uso de outros arquivos funciona em duas etapas: (1) define-se qual arquivo deve ser importado associando-o a um identificador que representa um novo tipo de dados usado para (2) referenciar o conteúdo desse arquivo ao longo do documento como mostra o exemplo da Figura 5.

```
// person.orderly
object {
    string name;
    integer age;
}

// family.orderly
import "person.orderly" as Person;

object {
    Person father;
    Person mother;
    array [Person] children;
}
```

Figura 5: Exemplo de referência a arquivos externos

O interpretador processa de forma recursiva todas as definições de *import* e por isso elas devem ser obrigatoriamente definidas no começo do arquivo para garantir a disponibilidade da referência. O token “as” não tem outro valor sintático que não facilitar a leitura da linha de importação dando maior clareza sobre o objetivo do identificador que a segue.

Como o identificador definido na linha de importação é usado para declarar um novo tipo de dados, não pode estar em conflito com os nomes já reservados para tipos como *string*, *array* e *boolean*.

3.2.2 Implementação de Dependência (Requires)

A implementação da extensão de dependência entre propriedades foi feita como proposta na Seção 3.1.3 mas com uma melhoria: a possibilidade de se definir mais de um valor possível em uma estrutura parecida com um literal de array.

As diferentes formas de se especificar dependências estão demonstradas na Figura 6. É importante notar que a implementação não faz a resolução de expressões não sendo aceito outro token além de “=” como por exemplo “<” e “>” para definir relação entre números. O motivo pelo qual essa forma não foi implementada é a ambiguidade gramatical de se usar esses dois tokens em uma definição que é encap-

sulada por esses mesmos tokens. Esse problema poderia ser resolvido trocando o delimitador da expressão, não feito nesse trabalho para minimizar o impacto na sintaxe da linguagem.

Essa extensão altera a forma como a especificação de dependência era interpretada na gramática original. Antes a definição era inversa, caso uma propriedade tivesse a especificação de dependência isso fazia com que a outra propriedade fosse obrigatória. Por exemplo, caso a propriedade atual exista, a propriedade dentro da definição de dependência também deve exisistir. Diferente do caso atual onde se a propriedade dentro da definição de dependência existir, a propriedade atual também deve existir.

```
object {
    integer previousSubmissions;
    // name so e obrigatorio se
    // previousSubmissions for igual a 0
    string name <previousSubmissions = 0>;

    string job?;

    // Caso job exista salary e obrigatorio
    number salary <job>;

    // Se job for physicist ou engineer
    // field e obrigatorio
    string field <job in ["physicist","engineer"]>
}*;
```

Figura 6: Exemplo de uso da definição de dependência

3.2.3 Alterações em Arrays

Arrays originalmente dependiam de uma definição obrigatória de qual tipo eles aceitam. Essa restrição foi removida permitindo uma definição mais concisa de arrays. Antes era possível declarar um array de qualquer tipo definindo um array que permite o tipo *Any*. Isso pode ser feito de forma implícita como demonstram os exemplos da Figura 7.

Outra alteração foi a remoção de uma característica da linguagem que era a abstração de tuplas usando arrays. Antes podia-se definir

```
object {
    // Antes essa era a unica forma de
    // definir arrays de qualquer tipo
    array [Any] arrayAntes;

    // Isso pode agora ser abreviado
    // mantendo a legibilidade
    array [] arrayPermitidoAgora;

    // Anteriormente a defincao
    // abaixo descrevia tuplas e
    // agora isso define arrays que
    // aceitam strings e integers
    array {string; integer} formerTupleTyping;
}
```

Figura 7: Alterações na definição de Arrays

um array que aceitava diferentes tipos de dados de forma sequencial. Uma definição como *array* {*string, integer*} permite um array onde os elementos devem ser sequencias entre strings e inteiros como demonstra a Figura 8. Essa sintaxe agora é usada para descrever arrays que aceitam mais de um tipo de dado. Por consequência dessa alteração agora somente é permitido uma definição por tipo de dados para evitar abiguidade na interpretação. Por exemplo, caso seja definidos duas estruturas diferentes para objetos não há como saber qual das duas definições deve ser usada para validar o objeto encontrado.

A remoção de tuplas foi feita para evitar o uso indevido de arrays para definição de informação estruturada, essa alteração também visa diminuir o risco de uma interpretação errada em cima de uma estrutura comum em linguagens de programação. Objetos podem definir tipos estruturados de forma mais completa e com a restrição necessária. Essa alteração também tornou incompatível o marcador "*" que permitia propriedades adicionais ao final da tupla, essa possibilidade foi removida da gramática.

```
// A definicao abaixo
array {string , integer};

// Permitiria o seguinte array
["primeiro", 1, "segundo", 2, "terceiro", 3]

// Mas nao permitiria
["primeiro", "segundo", "terceiro", 1, 2, 3]

// A definicao abaixo que antes seria correta
// agora nao e mais valida
array {string {5,10}; string {,250}}
```

Figura 8: Definição de tuplas

3.2.4 Propriedades Opcionais

A posição do ponto de interrogação que marca a opcionalidade de uma propriedade foi alterada. Antes ele estava no final da definição após a especificação de valor padrão e dependência e está agora junto com o identificador. Essa mudança tende a facilitar a leitura da opcionalidade caso a definição seja longa uma vez que o usuário da linguagem tende a associar a definição do objeto a seu identificador.

Outra alteração relativa a opcionalidade está no fato de que toda propriedade que tiver definição de requisito ou valor padrão definido passa a ser, por consequência, opcional. Definir opcionalidade com o ponto de interrogação de forma explícita se torna redundante nesses casos.

4 INTERPRETADOR ORDERLY

Para esse trabalho estamos interessados em um interpretador que seja capaz de ler uma especificação Orderly e a use para validar um documento JSON. Para isso precisamos entender como estão organizados os componentes de um interpretador e qual será a configuração para nossa aplicação.

O componente mais importante do interpretador é o parser[1]. Esse componente é responsável por *entender* a estrutura léxica e sintática de uma entrada transformando-a em uma estrutura de dados discreta que pode ser manipulada por um aplicativo.

O parser é um software mas não precisa ser manualmente escrito reduzindo muito a complexidade da implementação e manutenção de um compilador e interpretador. Com o uso de um gerador de parser pode-se especificar usando uma DSL[2] qual o tipo de entrada é aceita, um compilador então se encarrega de fazer a geração do código necessário para o parsing. A escolha do gerador de parser usado para o interpretador Orderly está descrito na Seção 2.2.

Como o objetivo não é somente interpretar um documento Orderly mas usá-lo para validar dados JSON, vamos precisar de dois parsers. Um fará a interpretação do documento Orderly criando uma estrutura de árvore que abstrai o significado do documento. O outro usará essa árvore para validar um documento JSON. Na prática criamos dois componentes de software distintos para facilitar a manutenção desses parsers. Os componentes do interpretador estão representados no diagrama da Figura 9.

A implementação dos componentes do interpretador está documentada da seguinte forma: implementação do parser Orderly na Seção 4.1, implementação do parser JSON na Seção 4.2, implementação dos testes automáticos de aceitação e debug na Seção 4.3 e por fim o uso do interpretador como biblioteca no Capítulo 5.

---

[1] Um termo muito usado para descrever parsers em português é *analisador sintático*. Para manter a concisão e coerência do trabalho o termo utilizado será somente *parser*.

[2] Domain Specific Language são linguagens usadas em domínios específicos como especificação de componentes de hardware, configuração de aplicativos e outros. Orderly é uma DSL para especificação de estrutura dados JSON.

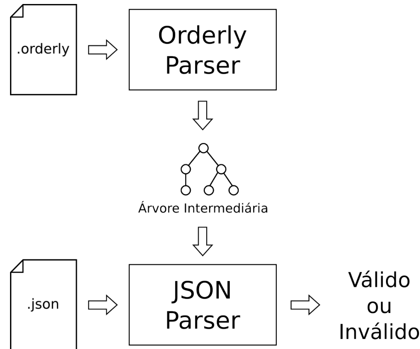

Figura 9: Componentes do Interpretador

4.1 PARSER ORDERLY

O parser Orderly é o componente de software responsável por interpretar um documento Orderly e criar uma árvore intermediária usada para validação de dados JSON. Ele também é responsável por validar se um documento orderly está correto em termos de sintaxe. Sua especificação é feita usando a linguagem de definição de parsers do ANTLR e nela está formalizada toda a sintaxe da linguagem orderly aceita pelo interpretador.

```
orderly_schema returns [JsonProperty rootProperty]
    : imports? unnamed_entry ';'?
      { $rootProperty = $unnamed_entry.property;}
    ;

import_statement
    : 'import' file=STRING 'as' id=IDENTIFIER ';'
      {this.addImport($id.text, $file.text);}
    ;

imports
    :   import_statement (import_statement)*
    ;
```

Os símbolos da gramática reproduzida acima mostram as definições de importação de arquivos descritas na Seção 3.2.1. O símbolo inicial *orderly_schema* retorna um objeto JsonProperty que representa

a gramática analisada.

Alguns métodos e atributos de apoio foram adicionados à classe do parser para facilitar as verificações semânticas. A principal delas é a verificação e armazenamento das importações de arquivos que são indexados por *indentificador* em um *HashMap<String, JsonProperty>*. Note que a decisão de representar a árvore intermediária como uma estrutura recursiva completa facilita muito o processo de reutilização de definições e abstração das propriedades.

4.1.1 Árvore Intermediária

A árvore intermediária gerada na primeira etapa de parsing é uma estrutura que abstrai os tipos da linguagem Orderly em objetos que podem ser aninhados usando as classes JSONObject e JSONArray. Essas classes foram todas implementadas extendendo JSONProperty e cada uma contém código adicional necessário para abstrair as propriedades específicas dos tipos que representam. O diagrama de classes na Figura 10 mostra como estão estruturados os elementos da árvore.

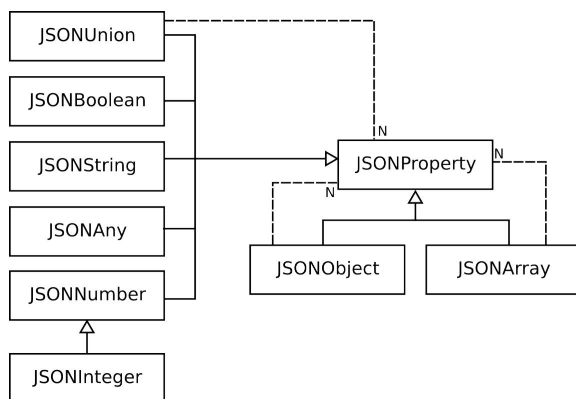

Figura 10: Classes da Árvore Intermediária

A classe abstrata JSONProperty é super-classe de todos os elementos da árvore intermediária. Nela estão implementados os métodos e atributos comuns a todos os tipos como valor padrão, referências (requires) e o nome da propriedade.

JSONArray é um importante componente de recursão na gramática e, como pode conter mais de um tipo de dados, foi implementado uma

estrutura de controle que indexa as declarações pelos tipos. Assim quando estiver fazendo o parsing de um array JSON o interpretador sabendo seu tipo pode pegar a declaração correta do array. No caso de números a verificação do parser JSON é feita abstraindo integers para números, essa tradução é transparente e não causa problemas devido a integer ser somente uma restrição sobre números.

4.1.1.1 JSONObject e Verificação de Referência

A classe JSONObject representa objetos e é responsável por fazer a checagem e resolução das dependências e também a verificação semântica de duplicação de identificadores e presença de propriedades obrigatórias.

A verificação de identificadores é feita usando o atributo *allMark*, esse atributo é uma lista que é preenchida ao longo do parsing com todos os identificadores encontrados. No momento da inserção caso um identificador já exista, um erro semântico é gerado informando a duplicidade na declaração de identificadores como mostrado a seguir.

```
tests/object/fail-duplicate-property.json
line 10:13 rule object failed predicate:
  {Duplicated property: string}?
```

Um atributo chamado *keyMark* é inicializado com todas as propriedades orderly não opcionais mais as propriedades que tem alguma dependência. Ao encontrar uma propriedade que está presente nessa lista ela é removida. Ao final do parsing do objeto os valores que sobram nessa lista são usados para verificar se as propriedades obrigatórias não foram negligênciadas. Caso as restrições não sejam respeitadas um erro semântico é gerado.

```
tests/object/fail-less-properties.json
line 7:1 rule object failed predicate:
  {Missing properties: stuff integer boolean }?
```

Caso uma propriedade encontrada não tenha sido declarada no objeto e o mesmo não permitir propriedades adicionais um erro semântico indica a presença de uma propriedade inexperada.

```
tests/object/fail-extra-properties.json
line 10:12 rule object failed predicate:
  {Unexpected property: extra}?
```

A verificação de dependência também acontece na mesma etapa da verificação de obrigatoriedade. Como descrita na Seção 3.2.2 a referência só faz sentido ser verificada caso a propriedade não tenha sido encontrada. Se o valor não foi encontrado *e* a restrição de dependência seja válida um erro semântico é gerado informando que a restrição de dependência não foi respeitada.

```
tests/requires/fail-simple-requires.json
line 5:1 rule object failed predicate:
  {Missing properties: salary }?
```

4.2 PARSER JSON

O parser JSON é usado para alcançar o objetivo principal do interpretador: validar um documento JSON usando como referência a árvore intermediária gerada a partir de um documento orderly. Ele faz isso iterando recursivamente sobre árvore intermediária e os dados JSON avaliando se a entrada está compatível com o que era esperado pelo parser.

O resultado do parsing orderly é uma propriedade JSON que é passada como atributo do parser JSON, essa propriedade é tratada como uma expectativa, isso é, a propriedade que o parser espera encontrar ao processar o primeiro símbolo JSON do documento. O código abaixo é um trecho da gramática ANTLR do parser JSON e mostra a implementação da definição de um valor JSON, símbolo principal da gramática. Note que as verificações semânticas são feitas com o auxílio de propriedades e métodos do próprio objeto da árvore intermediária.

```
jsonValue
  @init {
    JsonProperty expected = this.expectedProperty;
    expected.setInput(this.input);

    boolean isAny = this.expectedProperty instanceof JsonAny;
  }
  @after { this.expectedProperty = expected; }
  // String
  : { this.expectedProperty.allow(JsonString.class) }?
    {
      this.expectedProperty =
        resolvArrayContext(JsonString.class.getName());
```

```
      isAny = this.expectedProperty instanceof JsonAny;
    }
    string=STRING
    {
      isAny || JsonString.class.cast(this.expectedProperty)
                         .isValid($string.getText())
    }?
    {if(!isAny) {
      JsonString.class.cast(this.expectedProperty)
                .setValue($string.getText());
    }}
  // Object
  | { this.expectedProperty.allow(JsonObject.class) }?
    {
      this.expectedProperty =
        resolvArrayContext(JsonObject.class.getName());
      isAny = this.expectedProperty instanceof JsonAny;
    }
    jsonObject
  // Array
  | { this.expectedProperty.allow(JsonArray.class) }?
    {
      this.expectedProperty =
        resolvArrayContext(JsonArray.class.getName());
      isAny = this.expectedProperty instanceof JsonAny;
    }
    {if(!isAny) {
      JsonArray.class.cast(this.expectedProperty).setInside(true);
    }}
    jsonArray
    {if(!isAny) {
      JsonArray.class.cast(this.expectedProperty).setInside(false);
    }}
  ;
```

Esse parser, diferente do parser orderly, não tem um retorno para seu símbolo principal. A verificação do resultado é feita pegando o contador de erros do parser. Caso ele seja maior que 0 significa que houve erros no parsing e o documento é inválido. Os erros de gramática são atualmente impressos na saída padrão de erro do programa, comportamento padrão dos parsers gerados pelo ANTLR.

4.3 TESTES E DEBUGGING

Assim como no português, em uma gramática para uma linguagem de computador uma vírgula faz toda a diferença, uma alteração simples pode alterar o significado sintático de diversas partes da gramática mesmo que separando os trechos em macro símbolos e por isso é importante que se tenha um conjunto completo de testes para validar a gramática.

Para facilitar a validação das gramáticas Orderly e JSON desenvolvidas para o interpretador foi criado um script simples que procura em um diretório e executa diversos casos de teste (um para cada subdiretório) comparando o resultado. Após cada alteração da gramática os testes são executados novamente para garantir que nada do que já estava funcionando tenha parado de operar corretamente.

O script, reproduzido no Anexo C, faz o tratamento dos casos de teste da seguinte forma: em cada diretório de teste deve conter um arquivo .orderly contendo a gramática que se deseja testar e diversos arquivos .json para serem testados contra a gramática de referência. Caso o nome do arquivo .json inicie com “fail-” o resultado esperado deve ser um ou mais erros, do contrário uma saida normal sem erros é esperada.

Um exemplo de uma parte da saída desse script está reproduzida a seguir. A saída do script em um terminal unix é colorida para facilitar a leitura.

```
henrique@trantor:/tcc/jorderly$ ./runTests.sh typed-array

typed-array
===========

tests/typed-array/array.json
----------------------------

tests/typed-array/fail-below-range.json
---------------------------------------
tests/typed-array/fail-below-range.json
line 1:1 rule array failed predicate:
  {Array has less elements then it should}?

tests/typed-array/fail-over-range.json
--------------------------------------
```

```
tests/typed-array/fail-over-range.json
line 1:34 rule array failed predicate:
  {Array has more elements then it should}?
```

4.3.1 Debug Avançado da Gramática

Para casos onde é necessário um debug mais avançado sobre o que está sendo reconhecido pelo parser, o ANTLR possui uma facilidade de debug remoto que permite acompanhar em tempo de execução o que está sendo interpretado pelo parser. Para isso usa-se o ANTLR Works, uma IDE para criação de gramáticas com funcionalidades de debug e visualização avançadas. Isso se mostra muito útil na criação das definições da gramática ajudando a verificar se as expressões regulares e estruturas semânticas estão reconhecendo o texto da forma planejada facilitando a eliminação de erros pequenos.

Para fazer o debug usando esse recurso é necessário recompilar os arquivos da gramática com a opção “-debug”. Isso faz com que antes de iniciar uma operação de parsing o fluxo seja interrompido aguardando a conexão do debugger que assume o controle do fluxo de execução do parser. Uma amostra do debugging sendo feito na gramática Orderly está representada na Figura 11.

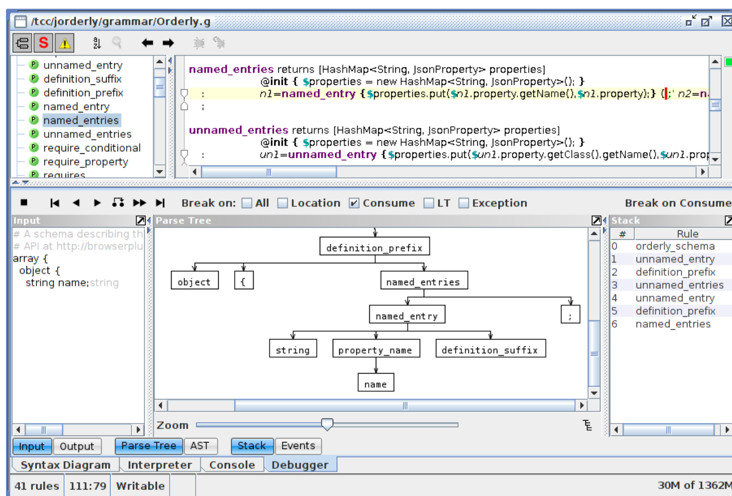

Figura 11: Debug Interativo do ANTLR Works. A imagem mostra a árvore de tokens gerada pelo parser permitindo visualizar se a interpretação ocorreu da forma que se espera.

5 USANDO O INTERPRETADOR

Para alcançar o objetivo de não depender de etapas prévias de compilação manual os parsers foram agrupados em uma API que integra seu uso. O seguinte código mostra como um usuário pode fazer uso da API para checar documentos JSON em seu código para fazer o parsing de um arquivo.

```
String grammarFile = "tests/import/grammar.orderly";
OrderlyParser parser = new OrderlyParser(grammarFile);

int errors = parser.parseFile("tests/import/import.json");
```

Podemos também usar outro método para fazer parsing de uma string contendo os dados JSON. Como a implentação do ANTLR é feita em cima de *streams* é possível com pouco esforço fazer a validação de um stream de rede reduzindo latência na análise. O exemplo abaixo mostra como uma string pode ser validada, espaços e quebras de linhas foram adicionados para legibilidade apesar de não serem sintaxe java válida.

```
jsonString = "{
    "father" : {
        "name" : "John",
        "age" : 50
    },
    "children" : [
        { "name" : "Henry", "age" : 22 },
        { "name" : "Gabriel", "age" : 23 }
    ]
}";

String file = "tests/import/grammar.orderly";
OrderlyParser parser = new OrderlyParser(file);

int errors = parser.parseString(jsonString);
```

6 CONCLUSÕES E TRABALHOS FUTUROS

O objetivo principal deste trabalho apresentado na Seção 1.1 é resolver os problemas discutidos na Seção 3.1 e para isso foi desenvolvido um interpretador Orderly que funciona em forma de biblioteca como projetado na Seção 4 atacando os problemas relacionados a necessidade de compilação para JSON Schema apresentado na Seção 3.1.1.

A gramática de orderly foi estendida para suportar novas construções em sua implementação como apresentado na Seção 3.2. Foram adicionadas a capacidade de referenciar outros documentos orderly facilitando o reuso de componentes descrito como problema na Seção 3.1.2 e a alteração no suporte a dependência de propriedades de objetos levantado na Seção 3.1.3.

Algumas outras alterações na gramática orderly também foram feitas e descritas na Seção 3.2. Estas com a motivação de melhorar a expressividade e legibilidade da linguagem orderly que também faz parte do objetivo principal deste trabalho.

Esse projeto não demandou somente um trabalho de extensão de algo existente, o fato de ter de implementar um interpretador desde a declaração de sua gramática e criação de estruturas de controle proporcionaram um grande aprendizado sobre como colocar na prática os conceitos aprendidos em diversas disciplinas do curso de Sistemas de Informação como Estrutura de Dados, Introdução a Compiladores e as diversas disciplinas de programação. A pesquisa e estudo necessário para conceber o projeto e procurar documentação de apoio assim como outros trabalhos que pudessem servir de base foram interessantes e ajudaram na descoberta de novas tecnologias e métodos para resolução de problemas computacionais.

Apesar do extenso componente de software produzido fazer um bom trabalho em endereçar os problemas apresentados, ainda há espaço para melhoria. Ao longo do desenvolvimento deste projeto diversas idéias foram surgindo sobre novas extensões além das melhorias que foram apresentadas e implementadas. Essas idéias são apresentadas nas próximas seções e podem servir como base para uma extensão deste trabalho enriquecendo as capacidades do software produzido.

6.1 DEPENDÊNCIA ATRAVÉS DE EXPRESSÕES

A implementação de dependência condicional apresentada na Seção 3.2.2 é feita com uma limitação de escopo onde uma dependência só é condicional sobre o valor de números e strings. Essas expressões poderiam combinar de forma mais elaborada mais de uma propriedade permitindo definições mais complexas. Abaixo estão alguns exemplos de como poderia ser implementada essa definição, o exemplo não leva em conta possíveis ambiguidades e problemas sintáticos.

```
object {
  boolean p1;
  string p2;
  number p3;

  string conditional requires p1 and (p2 = "value" or p3 > 5)
};
```

6.2 DECLARAÇÃO DE TIPOS E REFERÊNCIA INTERNA

Na definição de referência a arquivos externos, feita na Seção 3.2.1, optou-se por implementar a declaração da importação como a definição de um novo tipo de dados que é usado ao longo do documento Orderly. Essa implementação faz com que, mesmo que definindo tipos pequenos, seja necessário a criação de um arquivo adicional para ser importado. Isso poderia ser evitado se houvesse na linguagem a possibilidade de definição de um novo tipo dentro do mesmo arquivo. Isso poderia se dar como no exemplo a seguir.

```
def Person as object {
  string name;
  integer age
}

object {
  Person mother;
  Person father;
  array [Person] children?;
}
```

6.3 AÇÕES SEMÂNTICAS EM ORDERLY

Uma das extensões mais interessantes que podem ser adicionadas a orderly é a capacidade de definir ações semânticas em duas declarações permitindo assim que ela abstraia mais ainda a complexidade de se lidar com dados JSON. Isso poderia ter muita utilidade na construção de APIs REST que são orientadas a entrada e saída de suas chamadas.

Isso não precisa ser implementado de forma complexa como é feita pelo ANTLR pois isso demandaria que a gramática fosse compilada e tratada para garantir que as definições semânticas que forem anexadas ao documento não contenham erros. Uma solução elegante como a que foi aplicada ao GALS apresentado brevemente na Seção 2.2 pode ser aplicada a esse caso.

No GALS as definições semânticas são feitas inserindo marcadores numerados junto com as definições da gramatica criando meta-tokens. Quando esses tokens são interpretados a ação semântica adequada é chamada. Podemos aplicar esse conceito em Orderly mas aperfeiçoando um pouco para que ao invés de definir um numerador possamos usar identificadores de métodos que serão chamados em um objeto passado como parâmetro ao parser. O exemplo abaixo mostra como esses identificadores poderiam ser usados para inserir ações semânticas em uma especificação de JSON para uma api REST. Para dar uma melhor noção de contexto à aplicação prática vamos supor o método em questão como a adição (create) de pessoas em aplicação de agenda.

```
// Agenda entry
@initAgenda
array [
  object {
    string name;
    string address;
    string phone;
    @addPerson(name, address, phone)
  }
];
@saveAgenda
```

6.4 REPRESENTAÇÃO DISCRETA DO DOCUMENTO JSON

Uma extensão natural para esse trabalho é que o parser JSON faça, durante seu trabalho, a criação de uma estrutura de dados que represente os dados sendo validados. Com isso o interpretador pode ser usado para não só validar os dados mas para disponibilizá-los para o programa em um formato mais conveniente eliminando a necessidade de outra biblioteca para isso.

6.5 INTERPRETADOR PARA OUTRAS LINGUAGENS

Outro projeto interessante a derivar deste trabalho é a implementação desse mesmo interpretador em outras linguagens de programação. A mesma gramática desenvolvida para o ANTLR pode ser compilada em parsers de diversas linguagens, as ações semânticas e árvore intermediária devem porém ser re-implementadas nessa nova linguagem. Também é possível usar Jison (CARTER, 2012) para geração de um interpretador puramente em javascript.

REFERÊNCIAS

AHO, A. V. et al. *Compiladores: Princípios técnicas e ferramentas*. Segunda. [S.l.]: Pearson Education do Brasil, 2010. ISBN 9788588639249.

CAMPOS, R. L. R. *Introdução de Ações Semânticas na Ferramenta GALS*. dec. 2009. Http://projetos.inf.ufsc.br/arquivos_projetos/projeto_870/tcc.pdf. <http://projetos.inf.ufsc.br/arquivos_projetos/projeto_870/tcc.pdf>.

CARTER, Z. *Jison - Your friendly JavaScript parser generator!* nov. 2012. Http://zaach.github.com/jison/. <http://zaach.github.com/jison/>.

CROCKFORD, D. *The application/json Media Type for JavaScript Object Notation (JSON)*. IETF, jul. 2006. RFC 4627 (Informational). (Request for Comments, 4627). <http://www.ietf.org/rfc/rfc4627.txt>.

CROCKFORD, D. *JavaScript: The Good Parts*. [S.l.]: O'Reilly Media, Inc., 2008. ISBN 0596517742.

ECMA. *ECMAScript Language Specification*. December 1999. ECMA Standard 262, 3rd Edition. <http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf>.

FSF. *Bison - GNU parser generator*. nov. 2012. Http://www.gnu.org/software/bison/. <http://www.gnu.org/software/bison/>.

HILAIEL, L. *Orderly Documentation*. jun. 2012. Http://orderly-json.org/docs. <http://orderly-json.org/docs>.

MALESHKOVA, M.; PEDRINACI, C.; DOMINGUE, J. Investigating web apis on the world wide web. In: *Web Services (ECOWS), 2010 IEEE 8th European Conference on*. [S.l.: s.n.], 2010. p. 107–114.

PARR, T. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. First. [S.l.]: Pragmatic Bookshelf, 2007. (Pragmatic Programmers). ISBN 0978739256.

PARR, T. *ANTLR Parser Generator*. nov. 2012. Http://www.antlr.org/. <http://www.antlr.org/>.

ZYP, K.; COURT, G. *A JSON Media Type for Describing the Structure and Meaning of JSON Documents*. IETF, nov. 2010. 1-4 p. Http://tools.ietf.org/id/draft-zyp-json-schema-03.txt. <http://tools.ietf.org/id/draft-zyp-json-schema-03.txt>.

APÊNDICE A – Gramática do JSON

Grámatica JSON de acordo com a especificação ECMA 262 (ECMA, 1999).

```
JSONText :
  JSONValue

JSONValue :
  JSONNullLiteral
  JSONBooleanLiteral
  JSONObject
  JSONArray
  JSONString
  JSONNumber

JSONObject :
  { }
  { JSONMemberList }

JSONMember :
  JSONString : JSONValue

JSONMemberList :
  JSONMember
  JSONMemberList , JSONMember

JSONArray :
  [ ]
  [ JSONElementList ]

JSONElementList :
  JSONValue
  JSONElementList , JSONValue
```

APÊNDICE B – Gramática Orderly

Gramática Orderly como descrita no site oficial (HILAIEL, 2012). Essa gramática referência trechos da gramática do Anexo A.

```
orderly_schema
  unnamed_entry ";"
  unnamed_entry

named_entries
  named_entry ";" named_entries
  named_entry
  # nothing

unnamed_entries
  unnamed_entry ";" unnamed_entries
  unnamed_entry
  # nothing

named_entry
  definition_prefix property_name definition_suffix
  string_prefix property_name string_suffix

unnamed_entry
  definition_prefix definition_suffix
  string_prefix string_suffix

definition_prefix
  "integer" optional_range
  "number" optional_range
  "boolean"
  "null"
  "any"
  # a tuple-typed array
  "array"  "{" unnamed_entries "}" \
      optional_additional_marker optional_range
  # a simple-typed array
  "array"  "[" unnamed_entry "]" optional_range
  "object" "{" named_entries "}" \
      optional_additional_marker
  "union"  "{" unnamed_entries "}"

string_prefix
  "string" optional_range
```

```
string_suffix
  optional_perl_regex definition_suffix

definition_suffix
  optional_enum_values optional_default_value \
      optional_requires optional_optional_marker \
      optional_extra_properties
  # nothing

csv_property_names
  property_name "," csv_property_names
  property_name

optional_extra_properties
  "`" JSONObject "`"
  # nothing

optional_requires
  "<" csv_property_names ">"
  # nothing

optional_optional_marker
  "?"
  # nothing

optional_additional_marker
  "*"
  # nothing

optional_enum_values
  json_array
  # nothing

optional_default_value
  "=" json_value
  # nothing

optional_range
  "{" json_number "," json_number "}"
  "{" json_number "," "}"
```

```
  "{" "," json_number "}"
  "{" "," "}"
  # nothing

property_name
  json_string
  [A-Za-z_\-]+

optional_perl_regex
  "/" ([^/]|\/) "/" # a Perl 5 compatible re
  #nothing
```

APÊNDICE C – Script de Testes

Script usado para automatizar a execução de testes de aceitação e erro no interpretador desenvolvido neste trabalho. Exemplo do seu uso e uma descrição mais detalhada se encontram na Seção 4.3.

```
#!/bin/bash

# Text color variables
txtbld=$(tput bold)             # Bold
bldred=${txtbld}$(tput setaf 1) #  red
bldgrn=${txtbld}$(tput setaf 2) #  green
bldwht=${txtbld}$(tput setaf 7) #  white
txtrst=$(tput sgr0)             # Reset

TEST_PATH="tests"
JORDERLY="java -cp lib/*:dist/jorderly.jar \
          net.guax.jorderly.Main"

for testCase in `ls ${TEST_PATH}`; do

  if [ "$#" -ge 0 ] && [ "$testCase" == "$1" ] \
      || [ "$#" -eq 0 ];
    then

    echo
    echo -e "${bldgrn}${testCase}"
    s=$( printf "%${#testCase}s" ); echo "${s// /=}"
    echo -e "${txtrst}\n"

    for jsonFile in `ls ${TEST_PATH}/${testCase}/*.json`;
      do
      echo "${bldwht}${jsonFile}${txtrst}"
      s=$( printf "%${#jsonFile}s" ); echo "${s// /-}"
      echo -n ${bldred}
      if [[ "$jsonFile" == *"fail-"* ]]; then
          $JORDERLY ${TEST_PATH}/${testCase}/*.orderly \
              $jsonFile
          if [ $? = 0 ]; then
              echo -e "FAILURE, NO ERROR ON TEST!"
          fi
      else
          $JORDERLY ${TEST_PATH}/${testCase}/*.orderly \
              $jsonFile || echo -e "FAILURE ON TEST!"
```

```
      fi
      echo -n ${txtrst}
      echo -e "\n"
    done;
  fi
done;
```