Implementação de Extensibilidade Sintática com Tecnologia Adaptativa utilizando o JavaCC

A. S. Mignon

***Resumo*— Este trabalho objetiva relatar uma implementação de um analisador léxico e sintático de uma linguagem de programação procedimental simples, originalmente não extensível, associada a um mecanismo de extensão, que confere a tal linguagem a possibilidade de extensibilidade sintática. A linguagem de programação e o mecanismo de extensão foram originalmente propostos em [1]. A extensibilidade sintática da linguagem é implementada utilizando a tecnologia adaptativa. Para a geração do analisador léxico e sintático foi utilizada a ferramenta JavaCC.**

***Palavras chave*— JavaCC, extensibilidade, linguagem de programação, tecnologia adaptativa.**

I. Introdução

EM [1] os autores propõe um mecanismo de extensão de linguagens de programação imperativas originalmente não extensíveis. A associação deste mecanismo de extensão ao tipo de linguagem de programação citada permite a criação de novas construções sintáticas a partir de construções sintáticas pré-existentes na linguagem. Este tipo de mecanismo permite criar e utilizar construções sintáticas mais próximas ao domínio do problema em que se está trabalhando.

Este documento objetiva relatar uma implementação de um analisador léxico e sintático de uma linguagem de programação imperativa simples associada ao mecanismo de extensão citado. O objetivo deste mecanismo é permitir que a linguagem de programação apresente características de extensibilidade sintática. A tecnologia adaptativa [2] é utilizada para possibilitar que o reconhecedor original da linguagem seja capaz de incorporar o reconhecimento de novas construções sintáticas. Para a implementação do analisador léxico e sintático é utilizada a ferramenta JavaCC[4]. Esta ferramenta tem por objetivo gerar analisadores léxicos e sintáticos de linguagens de programação.

A Seção II apresenta sucintamente o trabalho na qual implementação foi baseada e como a tecnologia adaptativa é utilizada para implementar a extensibilidade sintática da linguagem. A Seção III apresenta a ferramenta JavaCC utilizada na implementação do analisador léxico e sintático. A Seção IV apresenta o relato da implementação realizada. Por fim, a Seção V apresenta conclusões e trabalhos futuros.

[4] Disponível em: <https://javacc.dev.java.net/>. Acesso em: 10 de dezembro de 2007.

II. Mecanismo de Extensibilidade Sintática

Esta seção baseia-se integralmente em [1]. Neste trabalho os autores propõem um mecanismo de extensão que, associado a linguagens de programação imperativas, originalmente não extensíveis, proporciona a linguagem características de extensibilidade sintática.

Uma linguagem de programação imperativa com características de extensibilidade sintática permite adicionar novas construções sintáticas a linguagem com o objetivo de representar abstrações não contidas em sua sintaxe. As novas construções sintáticas são especificadas a partir de termos existentes na linguagem. Estes termos podem tanto estar contidos na linguagem núcleo, quanto em alguma camada de extensão da linguagem já existente. A extensibilidade sintática permite que novas abstrações sejam adicionadas a linguagem, sem a necessidade de nenhuma alteração em seu compilador.

Um modo possível para permitir a inclusão de novas construções sintáticas em uma linguagem de programação imperativa, é oferecer alguma característica meta-lingüística a esta linguagem. A notação de Wirth [3] foi escolhida pelos autores para oferecer tais características. Este modo é utilizado pelo mecanismo de extensão para permitir a extensibilidade sintática da linguagem.

Permitir a especificação de novas construções sintáticas não é tudo que se precisa para a extensão de uma linguagem. É necessário também, expressar as extensões utilizando construções sintáticas já existentes na linguagem. A solução adotada para tal caso é tipificar a extensão e utilizar a semântica operacional clássica para interpretar cada nova construção sintática em relação a uma construção sintática existente: cada nova extensão é declarada como uma gramática livre de contexto, e o seu significado é declarado como um texto expresso como um programa usando a sintaxe básica da linguagem núcleo ou construções sintáticas previamente declaradas. Desta forma, o compilador é informado da nova construção a ser aceita, e também como exatamente deve traduzi-la em um nível de abstração mais baixo.

A. *O Mecanismo de Extensão*

Os autores apresentam uma proposta simplificada de um mecanismo de extensão, para ser associado a linguagens

de programação imperativas e originalmente não extensíveis. A proposta é apresentada através de uma gramática livre de contexto especificada em notação de Wirth modificada [4]. Esta gramática é dividida em duas partes: a primeira parte especifica uma linguagem de programação núcleo não extensível; a segunda parte especifica a proposta do mecanismo de extensão. As duas partes da gramática são apresentadas a seguir.

```
//* Linguagem Núcleo *//
PROG = "BEGIN" ( DECL \ ";" ) "START" ( COM \ ";" ) "END" .
DECL = "VAR" ( id \ "," ) ":" "INTEGER" | PROCEDURE | EXTENSION .
COM = LABEL ":" PROG | id := EXPARIT | "GOTO" LABEL |
      "IF" EXPARIT ( ">" | "=" | "<" | "<>" ) EXPARIT
         "THEN" PROG ( "ELSE" PROG | ε ) | PREVIOUSNTERM .
EXPARIT = (( id | int | CALL ) \ ( "+" | "-" | "*" | "/" )) .
CALL = id "(" ( id | int | CALL \ "," ) ")" .
PROCEDURE = "FUNCTION" id "(" ( id ":" "INTEGER" \ ";" )")"
                       ":" "INTEGER" ";"
                       "START" ( COM \ ";" ) "END" ";" .
LABEL = id .
```

```
//* Mecanismo de Extensão Proposto *//
PREVIOUSNTERM = ∅ .
EXTENSION = "DEFINE" NEWNTERM ":" "NEW" NTERM "AS" WIRTHMOD
                    "MEANING" PREVIOUSWIRTHMOD "ENDDEFINE" .
NTERM = "PROG" | "DECL" | "COM" | "EXPARIT" | "EXTENSION"
      | "LABEL" | "NTERM" | "CALL" | "PROCEDURE" | "NEWNTERM"
      | "WIRTHMOD" | "PREVIOUSWIRTHMOD" | PREVIOUSNTERM .
NEWNTERM = id .
WIRTHMOD = ((( TERM | NTERM | NEWNTERM | "ε"
          | "(" WIRTHMOD ( "\" WIRTHMOD | ε ) ")" )
             ( "#" int | ε ) \ ( "|" | ε ) ) .
PREVIOUSWIRTHMOD = ((( TERM | NTERM | "ε"
        | "(" PREVIOUSWIRTHMOD ( "\" PREVIOUSWIRTHMOD | ε ) ")" )
           ( "#" int | ε ) \ ( "|" | ε ) ) .
```

B. *O Uso da Tecnologia Adaptativa*

A simples associação da linguagem de programação com o mecanismo de extensão permite o reconhecimento apenas das construções sintáticas da linguagem núcleo e das construções sintáticas relacionadas ao mecanismo de extensão. Entretanto, o reconhecedor deve ser capaz também de reconhecer as novas construções sintáticas e incorporá-las a linguagem. Para que o reconhecedor comporte-se da maneira desejada, a tecnologia adaptativa é utilizada.

Ao reconhecedor existente, ações adaptativas são criadas, de modo que, quando o reconhecedor identificar uma nova construção sintática, a ação adaptativa crie um reconhecedor para a nova construção sintática e acrescente este novo reconhecedor ao reconhecedor da linguagem já existente. Além da alteração na estrutura do reconhecedor, a ação adaptativa insere a nova extensão da linguagem em um conjunto de extensões. Inicialmente, este conjunto de extensões é vazio.

C. *Um Exemplo de Extensão da Linguagem Núcleo*

Como exemplo de extensão da linguagem núcleo, propõe-se adicionar à linguagem a declaração de pré-condições [5]. Uma sintaxe sugerida é a construção de uma palavra "PRE" seguida por uma condição (no caso da linguagem utilizada, as condições podem ser definidas pela relação de duas expressões aritméticas). Para atingir o objetivo proposto no exemplo, o programa pode declarar a nova sintaxe desejada conforme apresentada seguir.

```
DEFINE PRECONDITION: NEW COM AS
   "PRE" EXPARIT # 1 ( "=" | "<" | ">" | "<>" )
   EXPARIT # 2
MEANING
   "IF NOT (" EXPARIT # 1 ( "=" | "<" | ">" | "<>" )
   EXPARIT # 2 ") THEN ERROR() ELSE"
ENDDEFINE;
...

FUNCTION divide (n: INTEGER; d: INTEGER): INTEGER;
   START
   PRECONDITION d <> 0;
   divide:=n/d
   END;
...
```

No trecho de programa apresentado, o denominador da divisão será checado automaticamente para verificar se o seu valor é diferente de zero toda vez que a função divide for chamada. Caso o denominador esteja associado com o valor zero um erro é reportado.

Em termos do pré-processador da linguagem, o comportamento deste pode ser identificado na geração de um código expandido conforme apresentado a seguir.

```
FUNCTION divide (n: INTEGER; d: INTEGER): INTEGER;
   START
   IF NOT ( d <> 0 ) THEN ERROR () ELSE divide:=n/d
   END;
```

III. A FERRAMENTA JAVACC

O JavaCC é um gerador de analisadores léxicos e sintáticos utilizado para a geração de compiladores de linguagens de programação do tipo LL{k}. Ele permite a especificação da gramática da linguagem de programação em uma notação próxima a EBNF. As definições léxicas e sintáticas da linguagem são descritas em um único arquivo.

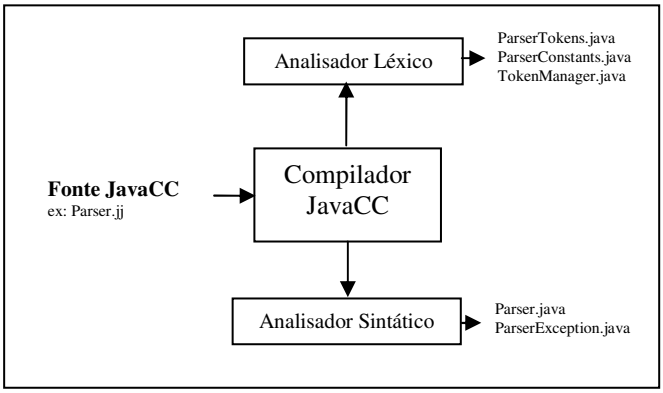

Fig. 1 Estrutura Básica do JavaCC.

A Fig. 1 apresenta a estrutura básica do JavaCC. Esta figura foi baseada em [6]. A especificação da gramática da linguagem de programação é descrita em um arquivo de extensão *.jj* (*Parser.jj*). Este arquivo é então submetido ao compilador do JavaCC. O compilador é responsável por gerar o analisador léxico e sintático da linguagem descrita. Os analisadores gerados são definidos em termos de classes Java. Para o analisador léxico são geradas três classes: *ParserTokens.java*, *ParserConstants.java* e *ParserTokenManager.java*. Para o analisador sintático são geradas duas classes: *Parser.java* e *ParserException.java*.

A. *Análise Léxica com o JavaCC*

O analisador léxico gerado pelo JavaCC é denominado *TokenManager*. O *TokenManager* é uma máquina de estados que move entre diferentes estados léxicos para a classificação dos *tokens*. Ele é utilizado para agrupar caracteres de um fluxo (*stream*) de entrada em *Tokens* de acordo com as regras léxicas especificadas. Cada regra léxica especificada no *TokenManager* é associada a expressões do tipo:

- SKIP: Descarta a expressão correspondida.
- MORE: Continua obtendo a próxima expressão para construir uma expressão maior.
- TOKEN: Cria um *token* utilizando a expressão correspondida e o envia ao analisador sintático.
- SPECIAL_TOKEN: Cria um *token* com a expressão correspondida e opcionalmente o envia ao analisador sintático.

O código a seguir apresenta um exemplo de uma especificação de *Tokens* em JavaCC. O tipo *SKIP* define que os espaços em branco e os caracteres de final de linha devem ser descartados. O tipo *TOKEN* define dois *tokens* da linguagem: o primeiro é o sinal de + e o segundo um número. Segundo a definição apresentada, um número pode conter um ou mais dígitos.

```
SKIP : { " " | "\n" | "\r" | "\r\n" }
TOKEN : {
      < PLUS : "+" >
  |   < NUMBER : (["0"-"9"])+ >
}
```

B. *Análise Sintática com o JavaCC*

O analisador sintático gerado pelo JavaCC é um analisador recursivo-descendente do tipo LL{k}. Este tipo de analisador utiliza um número *k* de *tokens lookahead* para gerar um conjunto de produções mutuamente exclusivas. Por padrão, o analisador sintático do JavaCC utiliza *k* igual a 1, entretanto, os desenvolvedores podem especificar o número do *lookahead* necessário para combinar corretamente as regras produção da linguagem [6].

O código a seguir apresenta um exemplo, em JavaCC, da especificação de uma regra de produção de uma gramática de uma linguagem de programação. Uma regra de produção é inicialmente definida por um nome, um tipo de retorno e, opcionalmente, parâmetros a serem passados para a regra. Após a definição do nome da regra, há um bloco delimitado por chaves que permite a inserção de código Java. Geralmente este bloco é utilizado para declaração de variáveis e chamada de métodos. Posteriormente, declaram-se regras de produção, contidas na gramática da linguagem, que estão associadas à regra de produção que está sendo definida. É possível associar código Java a cada uma das regras de produção declaradas.

```
void PROG() : {Token t;} {
  t=<NUMERO>
  {System.out.println(n.image.toString());}
      (<MAIS> <NUMERO>)*
      <EOF>
}
```

IV. UMA PROPOSTA DE IMPLEMENTAÇÃO

Esta seção relata a implementação da linguagem de programação livre de contexto e do seu mecanismo de extensão, apresentado na Seção II. Considera-se na implementação da linguagem apenas o seu analisador léxico e sintático. A ferramenta JavaCC, apresentada na Seção III, foi utilizada para gerar o analisador léxico e sintático proposto. A particular utilização desta ferramenta deve-se a familiaridade do autor com a mesma. Outro fator para a escolha foi considerar a implementação proposta em uma ferramenta não aderente a tecnologia adaptativa. O processo para a implementação da linguagem de programação e do mecanismo de extensão é descrito a seguir.

Inicialmente, a linguagem de programação núcleo foi especificada na ferramenta sem que o mecanismo de extensão fosse considerado. A única particularidade foi a inserção de um código escrito em linguagem Java para a geração de um *arquivo* de saída. O arquivo gerado não será gravado em disco, permanecendo, portanto, na memória enquanto o analisador sintático é executado. Este arquivo é responsável por armazenar o código-fonte do programa e a substituição das extensões encontradas. Posteriormente, foi especificada a parte da linguagem referente ao mecanismo de extensão proposto. Novamente, códigos em linguagem Java foram inseridos para auxiliar na geração do arquivo de saída.

À regra de produção denominada **EXTENSION** adiciona-se uma ação adaptativa posterior a análise sintática da extensão, escrita em linguagem Java, com o objetivo de colocar a definição da extensão em uma tabela. A implementação desta tabela foi realizada utilizando-se uma classe da linguagem Java denominada *HashMap*. Uma classe auxiliar, denominada *ExpansorExtensao*, foi criada e adicionada ao analisador sintático. Esta classe tem por objetivo, neste momento, armazenar a definição da extensão. A tabela armazena, portanto, o nome do novo termo criado e uma instância da classe *ExpansorExtensao*. A seguir é apresentado o código da regra de produção **EXTENSION** conforme especificado na ferramenta JavaCC. Nota-se que esta regra não grava nada no arquivo de saída que está sendo gerado pelo analisador sintático.

```
void EXTENSION(): {String n; String s, s1;}{
    <DEFINE> n=NEWNTERM() ":" <NEW> NTERM()
      <AS> s=WIRTHMOD()
    <MEANING> s1=PREVIOUSWIRTHMOD() <ENDDEFINE>
    {ExpansorExtensao ee =
      new ExpansorExtensao(n, s, s1);
     map.put(n, ee);}
}
```

O mecanismo de expansão de extensão, isto é, a substituição da nova construção sintática pelo seu significado, expresso por construções existentes na linguagem, foi implementado na regra de produção **PREVIOUSITEM**. Esta regra é acionada quando o analisador sintático encontra um identificador de acordo com a especificação do *Token* **<INDENT>**. A esta regra é associada uma ação adaptativa, descrita em linguagem Java, que verifica se o valor associado a este *Token* consta na tabela de extensões. Caso o identificador encontrado não conste na tabela de extensões, o

analisador sintático grava o *Token* consumido no arquivo de saída e continua o processamento do arquivo de entrada. Caso o valor conste na tabela, é obtida a instância da classe *ExpansorExtensao* associada ao identificador. Para efetuar a substituição do novo termo encontrado, o método *substituir* da classe *ExpansorExtensao* é acionado. Este método retorna uma *String* que contém o significado do termo encontrado. Esta *String* é então gravada no arquivo de saída que está sendo gerado pelo analisador sintático. Neste momento, o analisador sintático lê o restante do arquivo de entrada, grava os dados lidos no arquivo de saída, sem processá-los, e seu processamento é propositalmente interrompido. A necessidade da interrupção proposital do processamento do analisador sintático deve-se a limitações da ferramenta utilizada.

```
void PREVIOUSITEM() : {Token n;} {
      n=<IDENT>
      {if (map.containsKey(n.image.toString()))
{
         ExpansorExtensao ee =

      map.get(n.image.toString());
      saida.append(ee.substituir());
```

Após a interrupção do processamento do analisador sintático, este é novamente acionado, agora utilizando como arquivo de entrada o arquivo gerado no processamento anterior. Este processo continua até que o arquivo de entrada contenha somente notações sintáticas da linguagem núcleo e o analisador sintático aceite ou rejeite o arquivo que está sendo processado.

V. Conclusão

Este trabalho apresentou uma proposta de implementação de um analisador léxico e sintático de uma linguagem de programação imperativa simples, originalmente não extensível, associada a um mecanismo de extensão. O objetivo deste mecanismo é permitir que a linguagem de programação apresente características de extensibilidade sintática. A tecnologia adaptativa foi utilizada para possibilitar que o reconhecedor original da linguagem fosse capaz de incorporar o reconhecimento de novas construções sintáticas. Ações adaptativas foram utilizadas para alterar estruturalmente o reconhecedor da linguagem a cada nova construção sintática especificada.

A ferramenta JavaCC foi utilizada para a geração do analisador léxico e sintático da linguagem e do mecanismo de extensão. Entretanto, como a ferramenta não é aderente a tecnologia adaptativa, não se pode explorar ao máximo os recursos oferecidos por tal tecnologia. Para efeito de comparação, é recomendado que a implementação proposta fosse realizada em uma ferramenta de geração de analisadores léxicos e sintáticos aderente à tecnologia adaptativa.

Em trabalhos futuros pretende-se: criar um compilador, utilizando a tecnologia adaptativa, para a linguagem de programação e o mecanismo de extensão apresentados; efetuar uma comparação da implementação proposta em relação ao compilador citado; utilizar a tecnologia adaptativa em outros tipos de linguagens, como por exemplo, linguagens de transformação de modelos (ver ATL [7]).

Referências

[1] P. S. M. Silva and J. J. Neto, “An Adaptive Framework for the Design of Software Specification Languages,” in Proceedings of International Conference on Adaptive and Natural Computing Algorithms – ICANNGA 2005, Coimbra, March 21-23 2005.

[2] J. J. Neto, “Adaptive Rule-Driven Devices - General Formulation and Case Study,” in Implementation and Application of Automata: 6th International Conference, CIAA 2001, ser. Lecture Notes in Computer Science, B. W. Watson and D. Wood, Eds., vol. 2494. Pretoria, South Africa: Springer-Verlag, July 23-25 2001, pp. pp. 234–250.

[3] N. Wirth, “What can we do about the unnecessary diversity of notation for syntactic definitions?” Commun. ACM, vol. 20, no. 11, p. 822–823, 1977.

[4] J. J. Neto, “Contribuições à Metodologia de Construção de Compiladores,” Tese de Livre Docência, EPUSP, São Paulo, 1993.

[5] B. Meyer, Object-Oriented Software Construction, 2nd ed. New Jersey, USA: Prentice-Hall, 1997.

[6] G. Succi and R. W. Wong, “The Application of JAVACC to Develop a C/C++ Preprocessor,” SIGAPP Appl. Comput. Rev., vol. 7, no. 3, pp. 11–18, 1999.

[7] F. Jouault and I. Kurtev, “Transforming Models with ATL,” in Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005, 2005.