Um arcabouço para extensibilidade em linguagens de programação

P. R. M. Cereda e J. José Neto

***Abstract*—This paper aims at providing a support framework for programming languages through syntactic modifications at compilation time. Such modifications are available from an extension mechanism and allow an increase in expressive power. It is also desirable to use extensibility as a resource to provide special constructs targeting the project and implementation of programs with adaptative features.**

***Palavras-chave:*—Linguagens de programação, extensibilidade, adaptatividade**

I. Introdução

Dá-se o nome de *adaptatividade* à manifestação do fenômeno no qual um dispositivo modifica seu próprio comportamento espontaneamente, em resposta ao seu histórico de operação e aos dados de entrada [1], [2], [3]. Dados de entrada diferentes submetidos a um mesmo dispositivo adaptativo podem resultar, no final do processamento de tais dados e de acordo com diferentes séries de eventos do dispositivo, em configurações de comportamento e de topologia totalmente distintos. A adaptatividade tem como característica maior promover a extensão de formalismos já consolidados, aumentando seu poder de expressão [4].

A adaptatividade possui inúmeras aplicações computacionais nas mais diversas áreas de pesquisa. De acordo com José Neto [3], a generalidade resultante do modelo, a capacidade de aprendizado devida à característica de auto-modificação, bem como o poder de expressão faz dos dispositivos adaptativos uma alternativa muito atraente para expressar fatos complexos e manipular situações difíceis que surgem durante uma busca de soluções computacionais para problemas complexos.

Em particular, a área de pesquisa de linguagens de programação apresenta diversas contribuições extremamente significativas na utilização da tecnologia adaptativa em diversos níveis de abstração [5], [6], [7], [8], [9], [10]. Este artigo apresenta um arcabouço que ofereça extensibilidade em linguagens de programação através da especificação de construtos sintáticos definidos pelo usuário em uma metalinguagem de extensão. Como saída, o arcabouço permite a obtenção de um programa-objeto escrito em linguagem extensível a ser executado pelo sistema operacional ou de um programa-fonte que transforma as extensões sintáticas em comando válidos da linguagem base. A possibilidade de extensão de linguagens de programação pode proporcionar o projeto e implementação de construtos especiais para a escrita de programas com características adaptativas.

Este artigo está organizado da seguinte forma: a Seção II apresenta uma breve revisão bibliográfica dos assuntos relevantes ao arcabouço para extensibilidade em linguagens de programação, proposto neste artigo. A Seção III apresenta os subsídios para o projeto do arcabouço, tais como um conjunto de ações adaptativas para a geração de um analisador sintático, o mecanismo de tratamento de extensões para análise e injeção de novas construções sintáticas, e a organização estrutural proposta. A Seção IV apresenta algumas discussões em relação à proposta deste artigo, incluindo a necessidade de um protocolo de comunicação com o compilador da linguagem base. As considerações finais são apresentadas na Seção V.

Os autores podem ser contatados através dos seguintes endereços de correio eletrônico: paulo.cereda@usp.br e jjneto@usp.br.

II. Conceitos iniciais

Esta seção apresenta uma breve revisão bibliográfica dos assuntos relevantes ao arcabouço para extensibilidade em linguagens de programação, proposto neste artigo. São apresentados os conceitos referentes às linguagens extensíveis, autômatos de pilha estruturados, autômatos adaptativos, introdução à notação de Wirth como metalinguagem para descrição de linguagens de programação e sua interpretação, bem como uma breve discussão sobre a análise sintática e geração a partir da notação de Wirth.

A. Linguagens extensíveis

Linguagens extensíveis são linguagens de programação que permitem a adição ou alteração de construtos sintáticos ou a associação de novas formas sintáticas com semântica [11]. Os novos construtos podem incluir novas notações ou operações, estruturas de controle novas ou modificadas, ou até mesmo elementos provenientes de diferentes paradigmas de programação. Castro Junior [8] menciona os dois componentes de uma linguagem extensível e suas propriedades:

- *Componente léxico:* é descrito por expressões regulares da linguagem base, acrescidas das expressões regulares que compõem a linguagem que define o mecanismo de extensão.
- *Componente sintático:* para estender a sintaxe de uma linguagem base, deve-se fazer com que o compilador possa aceitar uma nova coleção de regras gramaticais, definidas em tempo de programação através de mecanismos declarativos incorporados à linguagem base.

A gramática da linguagem extensível é definida como a junção entre as gramáticas que definem o mecanismo de extensão e a linguagem base [8]. A partir de tais especificações, é possível obter um compilador para a linguagem extensível, de acordo com a Figura 1.

De acordo com a Figura 1, o compilador obtido incorpora as regras gramaticais da linguagem base e do mecanismo de extensão, construindo um reconhecedor apropriado para cada

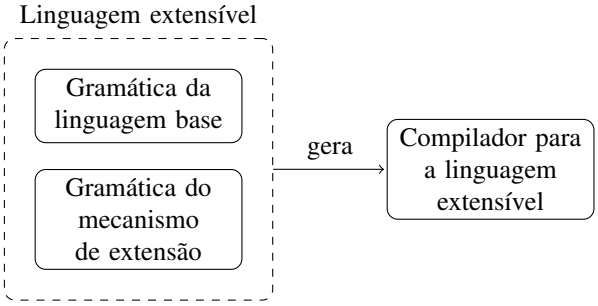

Figura 1. Obtenção do compilador para a linguagem extensível a partir das gramáticas da linguagem base e do mecanismo de extensão [8]

gramática e estabelecendo relacionamentos entre elas, quando aplicáveis.

A Figura 2 apresenta, em linhas gerais, o processo de obtenção do código-objeto para um programa P_0 escrito em uma linguagem extensível L_{E_0}, que consiste em um código-fonte escrito em uma linguagem base L_{base} acrescido de extensões sintáticas E_0 definidas pelo usuário, além da obtenção de um compilador para a nova linguagem L_{E_0}.

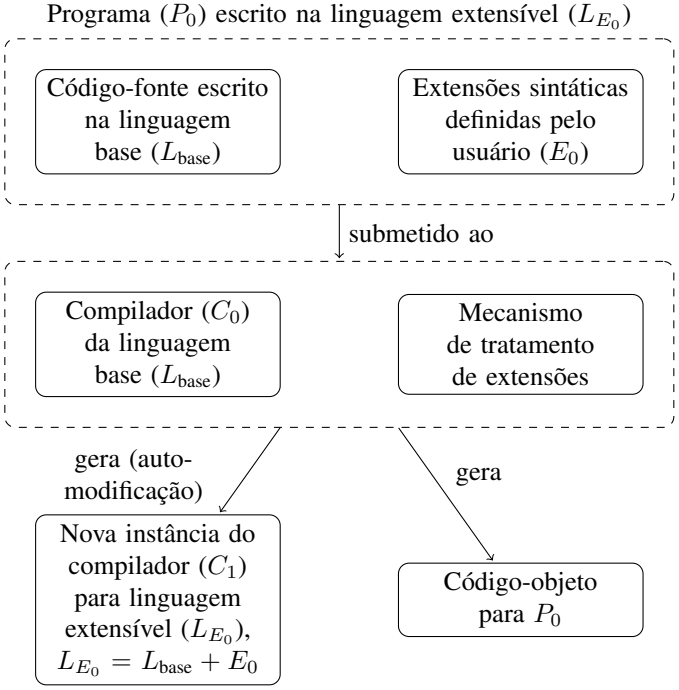

Figura 2. Obtenção do código-objeto para um programa escrito em L_{E_0} e do compilador C_1 que incorpora as novas construções sintáticas definidas pelo usuário (E_0) [8].

De acordo com a Figura 2, é possível notar que a nova linguagem L_{E_0} permite, por sua vez, o acréscimo de novas extensões sintáticas, tornando-se, portanto, a linguagem base de um novo processo de extensibilidade. Além disso, é possível utilizar-se deste conceito para projetar extensões que permitam a auto-modificação de código em tempo de execução, tornando a linguagem obtida propícia para a escrita de programas com características adaptativas.

Para o tratamento da linguagem que suporte adaptatividade por meio de extensão, de acordo com Castro Junior [8], é necessário projetar uma biblioteca de ações adaptativas para gerar as novas extensões em um formato que seja compatível com o ambiente de execução proposto para a linguagem base, e uma biblioteca de ações adaptativas para incorporar as extensões obtidas à nova instância da linguagem base.

Em [8], Castro Junior apresenta e discute os aspectos relacionados com o projeto de linguagens de programação adaptativa e introduz uma linguagem imperativa simplificada de programação denominada *MINLA* e sua linguagem de extensão, denominada *EXTLA*, definindo um conjunto de primitivas para o mapeamento de suas instruções na forma de construtos adaptativos. Adicionalmente, um ambiente de execução fundamentado no autômato adaptativo [1] é introduzido, definindo a estrutura básica da máquina de execução de tais programas, e implementando as seguintes rotinas [8]:

1) Carga do programa para a memória da máquina, cujo segmento de código pode ser representado por um estado do autômato que aponta para o estado correspondente à primeira instrução do respectivo código.
2) Busca da próxima instrução a ser executada no segmento de código, seguido de sua decodificação e execução.

Em [8], o autor ainda destaca que a implementação de um ambiente de execução adequado é muito vantajosa do ponto de vista experimental, uma vez que os comandos para a auto-modificação de código serão tratados sem a necessidade de grandes esforços para a adequação do ambiente de execução.

B. Autômato de pilha estruturado

O autômato de pilha estruturado [1] é um tipo de autômato de pilha formado por um conjunto de autômatos finitos mutuamente recursivos, também chamados de *sub-máquinas*. Diferentemente do autômato de pilha tradicional, a pilha tem a finalidade exclusiva de armazenar estados de retorno a cada chamada de uma sub-máquina. As chamadas e retornos consistem em transferir o controle entre uma sub-máquina e outra; essa transição consiste em utilizar o símbolo de entrada apenas para a tomada de decisão do autômato em relação a qual transição executar, sendo o tal símbolo consumido na próxima transição [1], [2].

Formalmente, um autômato de pilha estruturado M pode ser definido como $M = (Q, A, \Sigma, \Gamma, P, Z_0, q_0, F)$, onde Q é o conjunto finito não-vazio de estados, A é um conjunto de sub-máquinas, definidas a seguir, Σ é o alfabeto do autômato, correspondendo ao conjunto finito não-vazio dos símbolos de entrada, Γ é o conjunto finito não-vazio dos símbolos de pilha, armazenados na memória auxiliar do autômato, P é a relação de transição de estados, $q_0 \in Q$ é o estado inicial (da primeira sub-máquina), Z_0 é o símbolo marcador de pilha vazia, e $F \subseteq Q$ é o conjunto dos estados de aceitação do autômato (da primeira sub-máquina) [1].

Uma sub-máquina $a_i \in A$ é definida como um autômato finito tradicional, da forma $a_i = (Q_i, \Sigma_i, P_i, q_{i,0}, F_i)$, onde $Q_i \subseteq Q$ é o conjunto de estados de a_i, $\Sigma_i \subseteq \Sigma$ é o conjunto de símbolos de entrada de a_i, $q_{i,0}$ é o estado de entrada da sub-máquina a_i, $P_i \subseteq P$ é a relação de transição de estados de a_i, e $F_i \subseteq F$ é o conjunto de estados de retorno da sub-máquina.

A relação de transição P é definida como $P \subseteq \Gamma \times Q \times \Sigma \times \Gamma \times Q$, na forma $(\gamma g, e, s\alpha) \rightarrow (\gamma g', e', \alpha)$, onde e, e' são os estados corrente e de destino, respectivamente, s é o símbolo consumido, α é o restante da cadeia de entrada, g é o topo da pilha, g' é o novo topo da pilha, e γ é o restante da pilha. Uma configuração é um elemento de $Q \times \Sigma^* \times \Gamma^*$, e uma relação entre configurações sucessivas $\vdash$ é definida como:

- *Consumo de símbolo:* $(q, \sigma w, uv) \vdash (p, w, xv)$, com $p, q \in Q$, $u, x \in \Gamma$, $v \in \Gamma^*$, $\sigma \in \Sigma \cup \{\epsilon\}$, $w \in \Sigma^*$, se σ foi consumido pelo autômato, $x = u$, e $(\gamma, q, \sigma\alpha) \rightarrow (\gamma, p, \alpha) \in P$.
- *Chamada de sub-máquina:* $(q, w, uv) \vdash (r, w, xv)$, com $q, r \in Q$, $u \in \Gamma$, $v, x \in \Gamma^*$, $w \in \Sigma^*$, $x = pu$, com chamada da sub-máquina R, estado inicial r, retorno em p, e $(\gamma, q, \alpha) \rightarrow (\gamma p, r, \alpha) \in P$.
- *Retorno de sub-máquina:* $(q, w, uv) \vdash (p, w, v)$, com $p, q \in Q$, $u, x \in \Gamma$, $v \in \Gamma^*$, $w \in \Sigma^*$, $u = p$, com retorno de sub-máquina para p, e $(\gamma g, q, \alpha) \rightarrow (\gamma, g, \alpha) \in P$.

A linguagem reconhecida por um autômato de pilha estruturado M é dada por $L(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (f, \epsilon, Z_0), f \in F\}$.

C. Autômato adaptativo

O autômato adaptativo [1] é um extensão do formalismo do autômato de pilha estruturado (Subseção II-B) que permite o reconhecimento de linguagens recursivamente enumeráveis. O termo *adaptativo*, neste contexto, pode ser definido como a capacidade de um dispositivo em alterar seu comportamento de forma espontânea. Um autômato adaptativo, portanto, tem como característica a possibilidade de alterar sua própria topologia durante o processo de reconhecimento de uma dada cadeia, em resposta a algum estímulo [2]. A possibilidade de alteração do autômato ocorre através da utilização de ações adaptativas, que lidam com situações esperadas, mas ainda não consideradas, detectadas na cadeia submetida para reconhecimento pelo autômato [12].

Ao executar uma transição que contém uma ação adaptativa associada, o autômato sofre mudanças, obtendo-se uma nova configuração. Para a aceitação de uma determinada cadeia, o autômato percorrerá um caminho em um espaço de autômatos; haverá, portanto, um autômato E_0 que iniciará o reconhecimento de uma cadeia w, autômatos intermediários E_i que serão criados ao longo do reconhecimento, e um autômato final E_n que corresponde ao final do reconhecimento de w. Em outras palavras, seja a cadeia $w = \alpha_0\alpha_1 \ldots \alpha_n$; o autômato M descreverá um caminho de autômatos $\langle E_0, \alpha_0\rangle \rightarrow \langle E_1, \alpha_1\rangle \rightarrow \ldots \rightarrow \langle E_n, \alpha_n\rangle$, onde E_i representa um autômato correspondente à aceitação da sub-cadeia a_i.

Formalmente, um autômato adaptativo M pode ser definido como $M = (Q, A, \Sigma, \Gamma, P, Z_0, q_0, F, E, \Phi)$, onde Q é o conjunto finito não-vazio de estados, A é um conjunto de sub-máquinas, definidas a seguir, Σ é o alfabeto do autômato, correspondendo ao conjunto finito não vazio dos símbolos de entrada, Γ é o conjunto finito não-vazio de símbolos de pilha, armazenados na memória auxiliar do autômato, P é a relação de transição de estados, $q_0 \in Q$ é o estado inicial (da primeira sub-máquina), Z_0 é o símbolo marcador de pilha vazia, $F \subseteq Q$ é o conjunto dos estados de aceitação do autômato (da primeira sub-máquina), E representa o modelo de autômato utilizado (inicialmente, o reconhecimento inicia-se no autômato E_0 do caminho de autômatos), e Φ é o conjunto de funções adaptativas, aplicáveis às transições [1], [2].

Uma sub-máquina $a_i \in A$ do autômato adaptativo M é definida como $a_i = (Q_i, \Sigma_i, P_i, q_{i,0}, F_i, \Phi_i)$, onde $Q_i \subseteq Q$ é o conjunto de estados de a_i, $\Sigma_i \subseteq \Sigma$ é o conjunto de símbolos de entrada de a_i, $q_{i,0}$ é o estado de entrada da sub-máquina a_i, $P_i \subseteq P$ é a relação de transição de estados de a_i, $F_i \subseteq F$ é o conjunto de estados de retorno da sub-máquina a_i, e $\Phi_i \subseteq \Phi$ é o conjunto de funções adaptativas aplicáveis.

A relação de transição P é definida como $P \subseteq (\Gamma \times Q \times \Sigma \times (\Phi \cup \{\epsilon\})) \times (\Gamma \times Q \times \Sigma \times (\Phi \times \{\epsilon\}))$, na forma $(\gamma g, e, s\alpha), \mathcal{B} \rightarrow (\gamma g', e', s'\alpha), \mathcal{A}$, onde e, e' são os estados corrente e de destino, repectivamente, s é o símbolo consumido, α é o restante da cadeia de entrada, g é o topo da pilha, g' é o novo topo da pilha, γ é o restante da pilha, $\mathcal{B}$ é uma função adaptativa anterior, e $\mathcal{A}$ é uma função adaptativa posterior. Uma configuração para um autômato adaptativo é um elemento de $E \times Q \times \Sigma^* \times \Gamma^*$, e uma relação entre configurações sucessivas $\vdash$ é definida como:

- *Consumo de símbolo:* $(E_i, q, \sigma w, uv) \vdash (E_i, p, \sigma' w, xv)$, com $p, q \in Q$, $u, x \in \Gamma$, $v \in \Gamma^*$, $\sigma, \sigma' \in \Sigma \cup \{\epsilon\}$, $w \in \Sigma^*$, se σ foi consumido pelo autômato, $x = u$, e $(\gamma, q, \sigma\alpha) \rightarrow (\gamma, p, \sigma'\alpha) \in P$.
- *Chamada de sub-máquina:* $(E_i, q, w, uv) \vdash (E_i, r, w, xv)$, com $q, r \in Q$, $u \in \Gamma$, $v, x \in \Gamma^*$, $w \in \Sigma^*$, $x = pu$, com chamada da sub-máquina R, estado inicial r, retorno em p, e $(\gamma, q, \alpha) \rightarrow (\gamma p, r, \alpha) \in P$.
- *Retorno de sub-máquina:* $(E_i, q, w, uv) \vdash (E_i, p, w, v)$, com $p, q \in Q$, $u, x \in \Gamma$, $v \in \Gamma^*$, $w \in \Sigma^*$, $u = p$, com retorno de sub-máquina para p, e $(\gamma g, q, \alpha) \rightarrow (\gamma, g, \alpha) \in P$.
- *Funções adaptativas:* $(E_i, q, \sigma w, uv) \vdash^{\mathcal{B},\mathcal{A}} (E_{i+2}, p, \sigma' w, u'v)$, indicando $(E_i, q, \sigma w, uv) \rightarrow^{\mathcal{B}} (E_{i+1}, q, \sigma w, uv) \vdash (E_{i+1}, p, \sigma' w, u'v) \rightarrow^{\mathcal{A}} (E_{i+2}, p, \sigma' w, u'v)$, com $p, q \in Q$, $u, u' \in \Gamma$, $v \in \Gamma^*$, $w \in \Sigma^*$, se $(\gamma u, q, \sigma\alpha), \mathcal{B} \rightarrow (\gamma u, p, \sigma'\alpha), \mathcal{A} \in P$.

Uma chamada de função adaptativa $\mathcal{F}_i$ assume a forma $\mathcal{F}_i(\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,m})$, onde cada $\tau_{i,j}$ representa um argumento passado à função. A declaração de uma função adaptativa $\mathcal{F}_i$ com m parâmetros consiste de um cabeçalho da forma $\mathcal{F}_i(\Theta_{i,1}, \Theta_{i,2}, \ldots, \Theta_{i,m})$ e um corpo contendo nomes (uma lista de identificadores que representam objetos no escopo da função, incluindo variáveis e geradores) e ações (lista de ações adaptativas elementares precedida por uma ação adaptativa inicial e seguida por outra final). A inicialização de uma função adaptativa preenche os geradores e os parâmetros passados e, a seguir, as ações são efetivamente aplicadas.

São definidos três tipos de ações adaptativas elementares que realizam testes no conjunto de regras ou modificam regras existentes (relação de transição de estados P), a saber:

1) *ação adaptativa elementar de inspeção:* as ação não modifica o conjunto de regras, mas permite a inspeção deste para a verificação de regras que obedeçam um certo padrão.

2) *ação adaptativa elementar de remoção:* a ação remove regras que correspondem a um determinado padrão do conjunto corrente de regras.
3) *ação adaptativa elementar de inclusão:* a ação insere uma regra que corresponde a um determinado padrão no conjunto corrente de regras.

A linguagem reconhecida por um autômato adaptativo M é dada por $L(M) = \{w \in \Sigma^* \mid (E_0, q_0, w, Z_0) \vdash^* (E_n, f, \epsilon, Z_0), f \in F\}$. A simplicidade e facilidade de entendimento do autômato adaptativo em relação à máquina de Turing (modelo clássico de reconhecimento de linguagens recursivamente enumeráveis) fazem com que sua utilização seja muito ampla [3].

D. Notação de Wirth

Em [13], Niklaus Wirth introduz uma metalinguagem para descrição de linguagens de programação, na tentativa de oferecer uma notação simplificada em relação às iniciativas existentes; tal metalinguagem ficou conhecida como *notação de Wirth* e apresenta as seguintes propriedades:

1) A notação apresenta uma distinção clara entre metassímbolos, símbolos terminais e símbolos não-terminais. Os metassímbolos existentes são: `=`, `.`, `(`, `)`, `[`, `]`, `{`, `}`, `|` e `"`. Um símbolo não-terminal é denotado por um identificador, isto é, uma letra seguida zero ou mais letras e dígitos (como uma definição de variável em uma linguagem de programação), enquanto o símbolo terminal é expresso por uma cadeia de caracteres entre aspas duplas.
2) Não há restrição quanto à utilização de metassímbolos como símbolos da linguagem sendo descrita. Em outras palavras, o metassímbolo `|`, por exemplo, difere do símbolo terminal `"|"`.
3) A notação evita o uso intenso de recursão para expressar repetições simples; dessa forma, existe um construto para iteração explícita.
4) Não há a necessidade da utilização do uso de um símbolo explícito que represente a cadeia vazia (como $\langle empty \rangle$ na notação BNF ou ϵ), pois já existem construtos que tratam dessa situação.
5) A notação é baseada no conjunto de caracteres ASCII.

A metalinguagem proposta por Wirth é apresentada na Listagem 1, com as palavras `identifier` denotando um símbolo não-terminal, `literal` denotando um símbolo terminal e `character` denotando um símbolo válido.

```
syntax = { production } .
production = identifier "=" expression "." .
expression = term { "|" term } .
term = factor { factor } .
factor = identifier | literal | "(" expression ")" |
         "[" expression "]" | "{" expression "}" .
literal = """" character { character } """" .
```

Listagem 1. Notação de Wirth, em sua forma original [13], expressa utilizando a própria notação que a define.

De acordo com [13], a repetição é denotada por chaves, isto é, `{ a }` representa ϵ `| a | aa | aaa |` ... (fecho de Kleene). Elementos opcionais são expressos através de colchetes, isto é, `[ a ]` representa `a` $\mid \epsilon$. Parênteses são utilizados para representar agrupamentos, isto é, `( a | b) c` representa `ac | bc`. Os símbolos terminais são expressos entre aspas; caso as aspas apareçam como símbolos literais, estas são duplicadas[1].

E. Analisador sintático

A análise sintática (também chamada de *parsing*) é o segundo estágio de um processo de três partes (Figura 3) que o compilador emprega para a análise do programa-fonte e geração do programa-objeto [14]. O analisador sintático trabalha com o programa-fonte transformado pelo analisador léxico; ao invés de lidar com um fluxo de caracteres, ele trabalha com um fluxo de tokens que contêm palavras associadas a uma categoria sintática (semelhante à sua classe gramatical). A partir destas informações, o analisador sintático deriva uma estrutura sintática para o programa, adequando as palavras em um modelo gramatical da linguagem especificada; se o analisador sintático reconhece o fluxo de tokens como válido, o programa-fonte é válido sintaticamente, isto é, foi escrito de acordo com as especificações da gramática da linguagem; caso contrário, o analisador acusa o erro e fornece outras informações de diagnóstico ao usuário. Quando o programa-fonte é válido, o analisador constrói um modelo concreto do programa para ser utilizado no terceiro estágio, isto é, na geração de código propriamente dita [14].

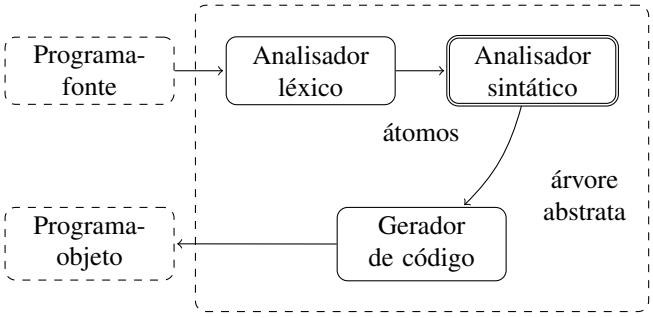

Figura 3. Organização de um compilador, tendo o analisador sintático como programa principal.

Uma linguagem de programação é, em geral, dependente de contexto. Entretanto, é comum tratá-la como livre de contexto, utilizando dispositivos reconhecedores de tais linguagens (como autômatos de pilha) e tratando a parte dependente de contexto como uma extensão semântica (o nome comum de tal verificação é *semântica estática*). A semântica estática é uma atividade realizada pelo analisador semântico para verificar a consistência do programa em termos das dependências existentes (por exemplo, se a expressão em um comando condicional realmente resolve para um valor lógico, se uma variável inteira recebe um valor inválido, como ponto flutuante, entre outros). O nome "semântica estática" é

[1] Algumas representações alternativas expressam aspas duplas literais como `"\""` ao invés de `""""`. O artigo original de Wirth mantém a opção pela duplicação das aspas.

inadequado porque a dependência de contexto é um fenômeno puramente sintático e não semântico; o analisador semântico acaba assumindo esse tratamento, além de suas funções usuais que compreendem a interpretação da linguagem [15], [2].

É possível gerar um analisador sintático a partir da notação de Wirth apresentada na Subseção II-D, obtendo-se um autômato de pilha estruturado que reconheça sentenças válidas da gramática especificada [15]. É imperativo que a gramática seja preparada previamente, tal que:

1) a gramática esteja escrita na notação de Wirth; caso esta esteja em outra notação, convertê-la, e
2) seja resolvido o sistema de equações representado pela gramática, interpretando os não-terminais como variáveis e os terminais como constantes.

Em relação ao item 2, a resolução do sistema de equações representado pela gramática consiste, em linhas gerais, em:

a) agrupar as várias opções do mesmo não-terminal,
b) fatorar as auto-recursões à esquerda e à direita,
c) transformar as auto-recursões à esquerda e à direita em iterações,
d) determinar os não-terminais essenciais, pesquisando-os a partir da raiz da gramática, de acordo com as dependências impostas pela gramática, e
e) eliminar, por substituição, os demais não-terminais.

Após o tratamento da gramática, é possível obter o analisador sintático utilizando o autômato de pilha estruturado apresentado na Figura 4; as ações semânticas associadas às transições estão definidas na Tabela I.

Tabela I
AÇÕES SEMÂNTICAS ASSOCIADAS ÀS TRANSIÇÕES DO AUTÔMATO DE PILHA ESTRUTURADO DA FIGURA 4.

Transição	Nome da ação semântica	Algoritmo
$(1, \texttt{'não-terminal'}) \rightarrow 2$	criar nova sub-máquina	1
$(2, \texttt{'='}) \rightarrow 3$	novo escopo	2
$(4, \texttt{'.'}) \rightarrow 5$	fechar escopo	3
$(5, \texttt{'não-terminal'}) \rightarrow 2$	criar nova sub-máquina	1
$(6, \texttt{'não-terminal'}) \rightarrow 7$	nova transição	4
$(6, \texttt{'terminal'}) \rightarrow 7$	nova transição	4
$(6, \texttt{'}\epsilon\texttt{'}) \rightarrow 7$	nova transição	4
$(7, \texttt{'não-terminal'}) \rightarrow 7$	nova transição	4
$(7, \texttt{'terminal'}) \rightarrow 7$	nova transição	4
$(7, \texttt{'}\epsilon\texttt{'}) \rightarrow 7$	nova transição	4
$(6, \texttt{'('}) \rightarrow 8$	novo escopo	2
$(6, \texttt{'['}) \rightarrow 10$	abre colchetes	5
$(6, \texttt{'\{'}) \rightarrow 12$	abre chaves	6
$(7, \texttt{'('}) \rightarrow 8$	novo escopo	2
$(7, \texttt{'['}) \rightarrow 10$	abre colchetes	5
$(7, \texttt{'\{'}) \rightarrow 12$	abre chaves	6
$(9, \texttt{')'}) \rightarrow 7$	fecha escopo	3
$(11, \texttt{']'}) \rightarrow 7$	fecha escopo	3
$(13, \texttt{'\}'}) \rightarrow 7$	fecha escopo	3
$(7, \texttt{'\|'}) \rightarrow 6$	adiciona opção	7

Algoritmo 1 Ação semântica de criação de nova sub-máquina

ação semântica *criar nova sub-máquina*

```
pilha.esvaziar()
estado corrente := 0
contador := 1
```

fim da ação semântica

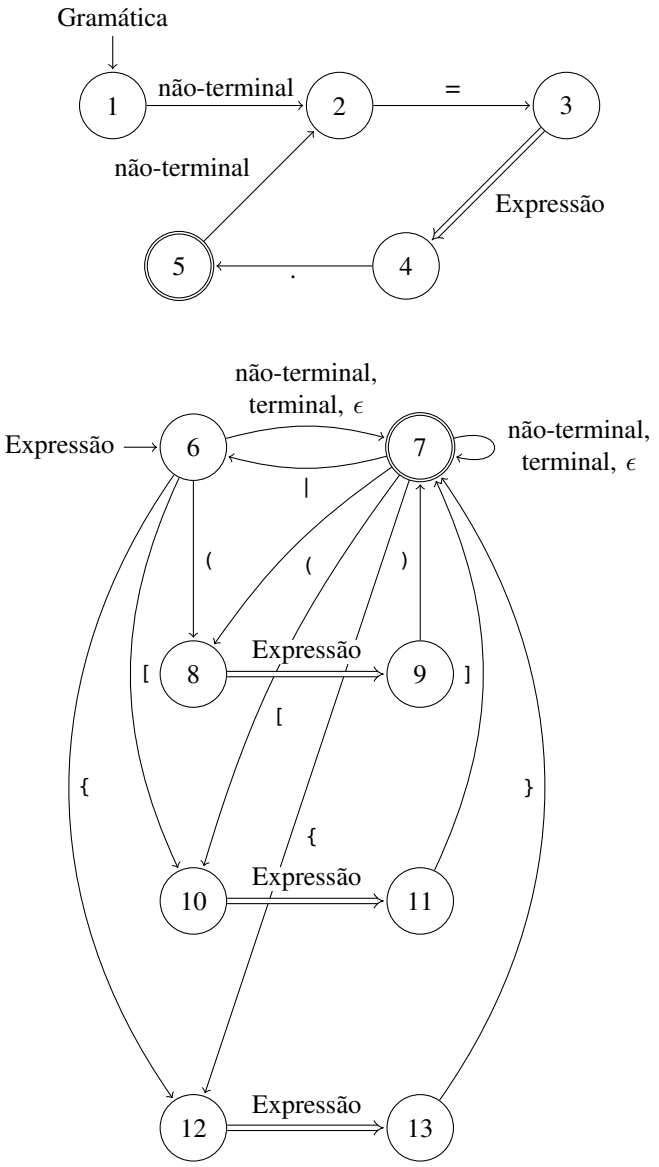

Figura 4. Autômato de pilha estruturado para geração do analisador sintático a partir de uma gramática escrita em notação de Wirth.

Algoritmo 2 Ação semântica de novo escopo

ação semântica *novo escopo*

```
pilha.empilhar(par(estado corrente, contador))
contador := contador + 1
```

fim da ação semântica

Algoritmo 3 Ação semântica de fechamento de escopo

ação semântica *fechar escopo*

```
transições += transição(estado corrente, ϵ,
pilha.topo().segundoElemento())
estado corrente := pilha.topo().segundoElemento()
pilha.desempilhar()
```

fim da ação semântica

Algoritmo 4 Ação semântica de nova transição

ação semântica *nova transição*

```
transições += transição(estado corrente,
token.valor(), contador)
estado corrente := contador
contador := contador + 1
```

fim da ação semântica

Algoritmo 5 Ação semântica de abertura de colchetes

ação semântica *abre colchetes*

```
transições += transição(estado corrente, ϵ,
contador)
pilha.empilhar(par(estado corrente, contador))
contador := contador + 1
```

fim da ação semântica

Algoritmo 6 Ação semântica de abertura de chaves

ação semântica *abre chaves*

```
transições += transição(estado corrente, ϵ,
contador)
estado corrente := contador
pilha.empilhar(par(contador, contador))
contador := contador + 1
```

fim da ação semântica

Algoritmo 7 Ação semântica de adição de opção

ação semântica *adiciona opção*

```
transições += transição(estado corrente, ϵ,
pilha.topo().segundoElemento())
estado corrente := pilha.topo().primeiroElemento()
```

fim da ação semântica

Como exemplo, considere a gramática de uma brincadeira de roda muito comum em países de língua inglesa chamada *duck, duck, goose*[2], apresentada na Listagem 2; a gramática do texto usado na brincadeira de roda é muito simples: a palavra `duck` deve ocorrer pelo menos uma vez, e pode repetir-se n vezes até que a palavra `goose` ocorra (com $n \geq 0$).

```
program = "duck" { "duck" } "goose" .
```

Listagem 2. Gramática do texto usado na brincadeira de roda *duck, duck, goose* escrita em notação de Wirth.

A partir da gramática da Listagem 2 e utilizando-se o autômato de pilha estruturado da Figura 4, obtém-se o autômato de pilha estruturado apresentado na Figura 5, que faz a análise sintática e reconhece sentenças pertencentes à brincadeira de roda *duck, duck, goose*.

É importante notar que o autômato gerado através do autômato de pilha estruturado da Figura 4 é não-determinístico, devendo este ser convertido para sua forma determinística. A minimização também pode ocorrer, mas deve ser utilizada com cautela para que estados semelhantes mas com semânticas diferentes não sejam aglutinados em um só, comprometendo o estágio de geração de código.

[2]O Brasil possui uma versão similar conhecida como *jogo do lenço*.

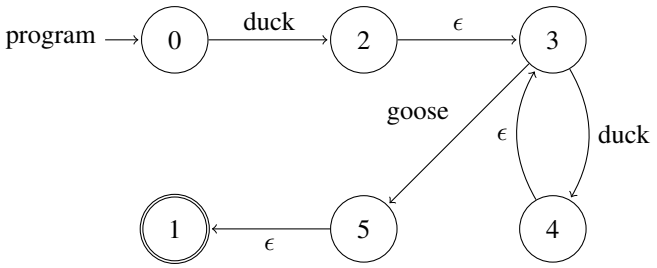

Figura 5. Autômato de pilha estruturado que reconhece sentenças pertencentes à brincadeira de roda *duck, duck, goose*.

III. Arcabouço para extensibilidade

Esta seção apresenta subsídios para o projeto de um arcabouço que ofereça extensibilidade em linguagens de programação, permitindo a adição ou a alteração de construtos sintáticos destas (Subseção II-A), tais como um conjunto de ações adaptativas para a geração automática de um analisador sintático a partir de uma gramática especificada utilizando-se a notação de Wirth (substituindo as ações semânticas discutidas na Subseção II-E), o mecanismo de tratamento de extensões para analisar e injetar novas construções sintáticas, e a organização do arcabouço proposto.

A. Ações adaptativas para geração do analisador sintático

O autômato adaptativo da Figura 6 foi inspirado no analisador sintático apresentado na Subseção II-E. Tal qual o autômato de pilha estruturado da Figura 4, o autômato proposto também constrói um autômato que reconheça sentenças válidas de uma linguagem especificada na notação de Wirth; entretanto, o autômato adaptativo altera sua topologia de forma que, ao terminar a leitura da cadeia de entrada (indicada pelo símbolo de terminação de arquivo $\langle \text{EOF} \rangle$), este poderá atuar como analisador sintático da linguagem recém-analisada, reconhecendo cadeias válidas de forma imediata. As ações adaptativas associadas às transições estão definidas na Tabela II.

Algoritmo 8 Função adaptativa $\mathcal{A}(p)$

função adaptativa $\mathcal{A}(p)$

geradores: $g_1^\star, g_2^\star$

$-(1, \texttt{'não-terminal'}) \rightarrow 2$

$+(\gamma, 1, \alpha) \rightarrow (\langle i, 6 \rangle \gamma, q_{p,g_1^\star}, \alpha)$

$+(\langle \text{Gramática}, 6 \rangle \gamma, g_2^\star, \alpha) \rightarrow (\gamma, q_{\text{Gramática},6}, \alpha)$

$+(g_1^\star, \tau_{\text{corrente}}) \rightarrow g_1^\star$

$+(g_2^\star, \tau_{\text{contador}}) \rightarrow g_2^\star$

fim da função adaptativa

É importante observar que, na especificação das ações adaptativas, a pilha sintática auxiliar originalmente utilizada nas ações semânticas do autômato de pilha estruturado da Figura 4 foi simulada através do encadeamento de estados auxiliares e

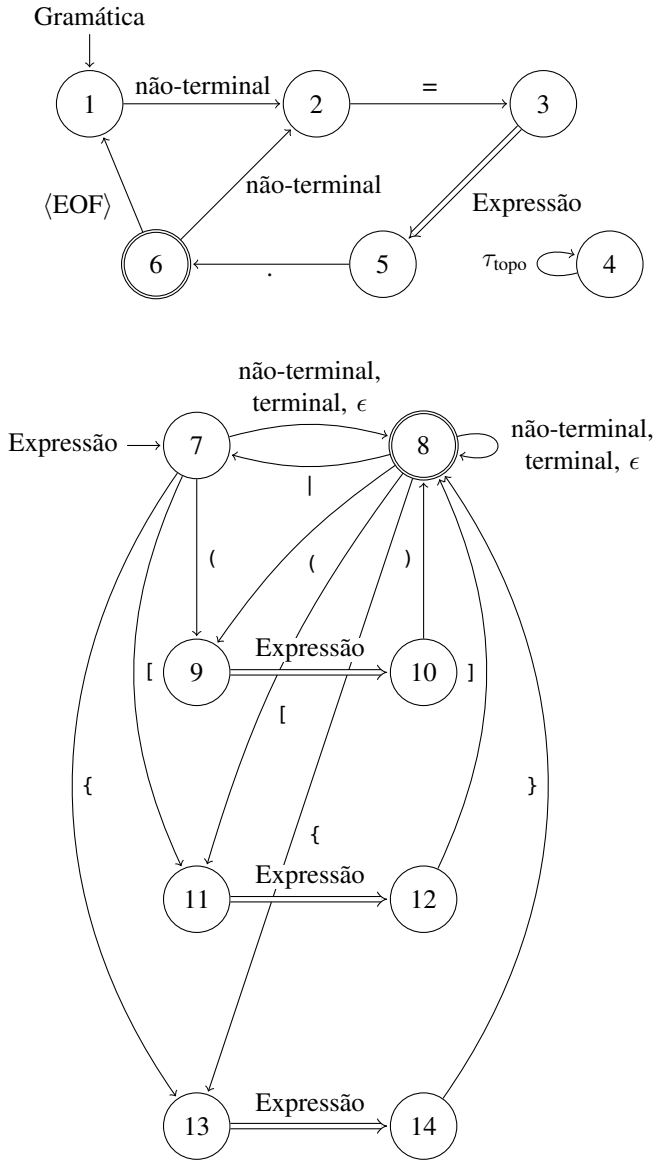

Figura 6. Autômato adaptativo generativo para realizar a análise léxica de sentenças escritas em um gramática definida através da notação de Wirth.

Algoritmo 9 Função adaptativa $\mathcal{B}()$

função adaptativa $\mathcal{B}()$
geradores: $g_1^\star, g_2^\star$
variáveis: $?a, ?b, ?c, ?v_1, ?v_2, ?v_3$

$?(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$?(?b, \tau_{\text{contador}}) \rightarrow ?v_2$
$?(?c, \tau_{\text{topo}}) \rightarrow ?v_3$
$-(?c, \tau_{\text{topo}}) \rightarrow ?v_3$
$+(?c, \tau_{\text{elemento}}) \rightarrow g_1^\star$
$+(g_1^\star, \tau_{\text{topo}}) \rightarrow g_1^\star$
$+(g_1^\star, \tau_{\text{primeiro}}) \rightarrow ?a$
$+(g_1^\star, \tau_{\text{segundo}}) \rightarrow ?b$
$-(?b, \tau_{\text{contador}}) \rightarrow ?v_2$
$+(g_2^\star, \tau_{\text{contador}}) \rightarrow g_2^\star$

fim da função adaptativa

Tabela II
AÇÕES ADAPTATIVAS ASSOCIADAS ÀS TRANSIÇÕES DO AUTÔMATO ADAPTATIVO GENERATIVO DA FIGURA 6.

Transição	Ação adaptativa	Algoritmo
$(1, \texttt{'não-terminal'}) \rightarrow 2$	$\cdot\mathcal{A}(\texttt{'não-terminal'})$	8
$(2, \texttt{'='}) \rightarrow 3$	$\cdot\mathcal{B}()$	9
$(5, \texttt{'.'}) \rightarrow 6$	$\cdot\mathcal{C}()$	10
$(6, \texttt{'não-terminal'}) \rightarrow 2$	$\cdot\mathcal{D}(\texttt{'não-terminal'})$	11
$(6, \langle\text{EOF}\rangle) \rightarrow 1$	$\cdot\mathcal{E}()$	12
$(7, \texttt{'não-terminal'}) \rightarrow 8$	$\cdot\mathcal{F}(\texttt{'não-terminal'})$	13
$(7, \texttt{'terminal'}) \rightarrow 8$	$\cdot\mathcal{G}(\texttt{'terminal'})$	14
$(7, \texttt{'}\epsilon\texttt{'}) \rightarrow 8$	$\cdot\mathcal{G}(\texttt{'}\epsilon\texttt{'})$	14
$(8, \texttt{'não-terminal'}) \rightarrow 8$	$\cdot\mathcal{F}(\texttt{'não-terminal'})$	13
$(8, \texttt{'terminal'}) \rightarrow 8$	$\cdot\mathcal{G}(\texttt{'terminal'})$	14
$(8, \texttt{'}\epsilon\texttt{'}) \rightarrow 8$	$\cdot\mathcal{G}(\texttt{'}\epsilon\texttt{'})$	14
$(7, \texttt{'('}) \rightarrow 9$	$\cdot\mathcal{B}()$	9
$(7, \texttt{'['}) \rightarrow 11$	$\cdot\mathcal{H}()$	15
$(7, \texttt{'\{'}) \rightarrow 13$	$\cdot\mathcal{I}()$	16
$(8, \texttt{'('}) \rightarrow 9$	$\cdot\mathcal{B}()$	9
$(8, \texttt{'['}) \rightarrow 11$	$\cdot\mathcal{H}()$	15
$(8, \texttt{'\{'}) \rightarrow 13$	$\cdot\mathcal{I}()$	16
$(10, \texttt{')'}) \rightarrow 8$	$\cdot\mathcal{C}()$	10
$(12, \texttt{']'}) \rightarrow 8$	$\cdot\mathcal{C}()$	10
$(14, \texttt{'\}'}) \rightarrow 8$	$\cdot\mathcal{C}()$	10
$(8, \texttt{'\|'}) \rightarrow 7$	$\cdot\mathcal{J}()$	17

Algoritmo 10 Função adaptativa $\mathcal{C}()$

função adaptativa $\mathcal{C}()$
variáveis: $?a, ?b, ?c, ?d, ?e, ?v_1, ?v_2$

$?(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$?(?b, \tau_{\text{topo}}) \rightarrow ?v_2$
$?(?b, \tau_{\text{primeiro}}) \rightarrow ?c$
$?(?b, \tau_{\text{segundo}}) \rightarrow ?d$
$+(?a, \epsilon) \rightarrow ?d$
$?(?e, \tau_{\text{elemento}}) \rightarrow ?b$
$-(?b, \tau_{\text{topo}}) \rightarrow ?v_2$
$-(?b, \tau_{\text{primeiro}}) \rightarrow ?c$
$-(?b, \tau_{\text{segundo}}) \rightarrow ?d$
$-(?e, \tau_{\text{elemento}}) \rightarrow ?b$
$+(?e, \tau_{\text{topo}}) \rightarrow ?e$
$-(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$+(?d, \tau_{\text{corrente}}) \rightarrow ?d$

fim da função adaptativa

Algoritmo 11 Função adaptativa $\mathcal{D}(p)$

função adaptativa $\mathcal{D}(p)$
geradores: $g_1^\star, g_2^\star$
variáveis: $?a, ?b, ?c, ?d, ?v_1, ?v_2, ?v_3, ?v_4$

$q_{p,0} \leftarrow g_1^\star$
$?(?a, \tau_{\text{topo}}) \rightarrow ?v_1$
$-(?a, \tau_{\text{topo}}) \rightarrow ?v_1$
$?(?b, \tau_{\text{elemento}}) \rightarrow ?v_2$
$-(?b, \tau_{\text{elemento}}) \rightarrow ?v_2$
$?(?c, \tau_{\text{primeiro}}) \rightarrow ?v_3$
$-(?c, \tau_{\text{primeiro}}) \rightarrow ?v_3$
$?(?d, \tau_{\text{segundo}}) \rightarrow ?v_4$
$-(?d, \tau_{\text{segundo}}) \rightarrow ?v_4$
$+(4, \tau_{\text{topo}}) \rightarrow 4$
$+(g_1^\star, \tau_{\text{corrente}}) \rightarrow g_1^\star$
$+(g_2^\star, \tau_{\text{contador}}) \rightarrow g_2^\star$

$+(\langle n, m\rangle\gamma, g_2^\star, \alpha) \rightarrow (\gamma, q_{n,m}, \alpha)$

fim da função adaptativa

Algoritmo 12 Função adaptativa $\mathcal{E}()$

função adaptativa $\mathcal{E}()$
variáveis: $?a, ?b, ?c, ?d, ?v_1, ?v_2, ?v_3, ?v_4$
$-(6, \langle\text{EOF}\rangle) \rightarrow 1$
$-(6, \texttt{'não-terminal'}) \rightarrow 2$
fim da função adaptativa

Algoritmo 13 Função adaptativa $\mathcal{F}(p)$

função adaptativa $\mathcal{F}(p)$
geradores: $g^\star$
variáveis: $?a, ?b, ?v_1, ?v_2$

$?(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$?(?b, \tau_{\text{contador}}) \rightarrow ?v_2$
$-(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$-(?b, \tau_{\text{contador}}) \rightarrow ?v_2$
$+(?b, \tau_{\text{corrente}}) \rightarrow ?b$
$+(g^\star, \tau_{\text{contador}}) \rightarrow g^\star$

$+(\gamma, ?a, \alpha) \rightarrow (\langle i, ?b\rangle\gamma, q_{p,a}, \alpha)$
fim da função adaptativa

Algoritmo 14 Função adaptativa $\mathcal{G}(p)$

função adaptativa $\mathcal{F}(p)$
geradores: $g^\star$
variáveis: $?a, ?b, ?v_1, ?v_2$

$?(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$?(?b, \tau_{\text{contador}}) \rightarrow ?v_2$
$+(?a, p) \rightarrow ?b$
$-(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$-(?b, \tau_{\text{contador}}) \rightarrow ?v_2$
$+(?b, \tau_{\text{corrente}}) \rightarrow ?b$
$+(g^\star, \tau_{\text{contador}}) \rightarrow g^\star$

fim da função adaptativa

Algoritmo 15 Função adaptativa $\mathcal{H}()$

função adaptativa $\mathcal{G}()$
geradores: $g_1^\star, g_2^\star$
variáveis: $?a, ?b, ?c, ?v_1, ?v_2, ?v_3$

$?(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$?(?b, \tau_{\text{contador}}) \rightarrow ?v_2$
$+(?a, \epsilon) \rightarrow ?b$
$?(?c, \tau_{\text{topo}}) \rightarrow ?v_3$
$-(?c, \tau_{\text{topo}}) \rightarrow ?v_3$
$+(?c, \tau_{\text{elemento}}) \rightarrow g_1^\star$
$+(g_1^\star, \tau_{\text{topo}}) \rightarrow g_1^\star$
$+(g_1^\star, \tau_{\text{primeiro}}) \rightarrow ?a$
$+(g_1^\star, \tau_{\text{segundo}}) \rightarrow ?b$
$-(?b, \tau_{\text{contador}}) \rightarrow ?v_2$
$+(g_2^\star, \tau_{\text{contador}}) \rightarrow g_2^\star$

fim da função adaptativa

Algoritmo 16 Função adaptativa $\mathcal{I}()$

função adaptativa $\mathcal{H}()$
geradores: $g_1^\star, g_2^\star$
variáveis: $?a, ?b, ?c, ?v_1, ?v_2, ?v_3$

$?(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$?(?b, \tau_{\text{contador}}) \rightarrow ?v_2$
$+(?a, \epsilon) \rightarrow ?b$
$?(?c, \tau_{\text{topo}}) \rightarrow ?v_3$
$-(?c, \tau_{\text{topo}}) \rightarrow ?v_3$
$+(?c, \tau_{\text{elemento}}) \rightarrow g_1^\star$
$+(g_1^\star, \tau_{\text{topo}}) \rightarrow g_1^\star$
$+(g_1^\star, \tau_{\text{primeiro}}) \rightarrow ?b$
$+(g_1^\star, \tau_{\text{segundo}}) \rightarrow ?b$
$-(?b, \tau_{\text{contador}}) \rightarrow ?v_2$
$+(g_2^\star, \tau_{\text{contador}}) \rightarrow g_2^\star$
$-(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$+(?b, \tau_{\text{corrente}}) \rightarrow ?b$

fim da função adaptativa

Algoritmo 17 Função adaptativa $\mathcal{J}()$

função adaptativa $\mathcal{I}()$
variáveis: $?a, ?b, ?c, ?d, ?v_1, ?v_2$

$?(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$?(?b, \tau_{\text{topo}}) \rightarrow ?v_2$
$?(?b, \tau_{\text{primeiro}}) \rightarrow ?c$
$?(?b, \tau_{\text{segundo}}) \rightarrow ?d$
$+(?a, \epsilon) \rightarrow ?d$
$-(?a, \tau_{\text{corrente}}) \rightarrow ?v_1$
$+(?c, \tau_{\text{corrente}}) \rightarrow ?c$

fim da função adaptativa

da utilização de sinalizadores especiais em cada execução das funções adaptativas, para fins didáticos. A utilização de uma pilha sintática auxiliar torna a implementação mais eficiente, reduzindo o número de estados criados e o número de ações elementares de inspeção no autômato.

B. Mecanismo de tratamento de extensões

O mecanismo de tratamento de extensões é um componente do arcabouço de extensibilidade que atua na manipulação de mudanças estruturais definidas pelo usuário, exigindo que o compilador modifique-se em tempo de compilação do programa-fonte, em resposta a tais mudanças requeridas sobre a linguagem original [8]. Uma metalinguagem é definida para o mecanismo de extensão, que realizará o mapeamento das novas construções sintáticas ao seu significado.

O mecanismo de tratamento de extensões analisa o processo de reconhecimento sintático da linguagem base, mapeando as novas construções sintáticas definidas na metalinguagem de extensão e disponíveis em seu componente sintático para o analisador original; assim, em tempo de compilação do programa-fonte, é possível alterar a topologia do analisador sintático original para acomodar as extensões definidas. A Figura 7 apresenta uma possível organização do mecanismo de tratamento de extensões, realizando a junção dos analisadores sintáticos das duas linguagens.

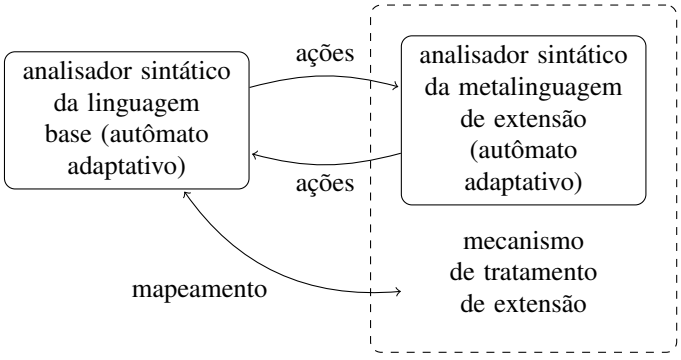

Figura 7. Uma possível organização do mecanismo de tratamento de extensões, realizando a junção dos analisadores sintáticos das duas linguagens.

De acordo com a Figura 7, o mecanismo de tratamento de extensões exerce um papel determinante para prover integração entre a linguagem básica e as extensões definidas pelo usuário; a adequação da sintaxe estendida em tempo de execução é

realizada através de ações adaptativas, que acomodam as novas construções no analisador sintático original.

Como exemplo, considere a gramática apresentada na Listagem 2 acrescida de ações semânticas, tal que `duck` $\rightarrow a_1$ e `goose` $\rightarrow a_2$; a entrada `duck duck goose`, portanto, gera a sequência de ações semânticas $\langle a_1, a_1, a_2 \rangle$. Suponha que a linguagem base seja estendida, através de uma definição utilizando a metalinguagem implementada pelo mecanismo de tratamento de extensões, para suportar a palavra `drake` ocorrendo zero ou mais vezes no início da sentença. Para fins ilustrativos, um exemplo de metalinguagem hipotética (fortemente inspirada em [8]) para suportar tal extensão é apresentada na Listagem 3.

```
DEFINE drake_command AS
   [ "drake":1 { "drake":1 } ] .
INJECT drake_command AT
   program.
MEANING
   1:
      a1
END
```

Listagem 3. Exemplo de metalinguagem hipotética, fortemente inspirada em [8], para suportar a nova palavra `drake` ocorrendo zero ou mais vezes no início da sentença.

A Figura 8 ilustra os dois analisadores sintáticos resultantes da geração através de ações adaptativas (Subseção III-A). É importante observar que algumas transições foram associadas a ações semânticas (note ainda que, neste exemplo particular, a extensão proposta faz uso tão somente dos construtos semânticos da linguagem base – admitindo que esta possui apenas as ações a_1 e a_2 disponibilizadas).

O mecanismo de tratamento de extensões, através de ações adaptativas, pode detectar a ocorrência da nova extensão em um programa-fonte (escrito na linguagem extensível) e realizar alterações no reconhecimento sintático para acomodar a nova situação, conforme ilustra a Figura 9.

De acordo com a Figura 9, o reconhecedor sintático já permite o tratamento de extensões definidas pelo usuário. Como exemplo, a sentença `drake drake duck duck goose` é corretamente aceita (linguagem extensível), resultando na sequência de ações semânticas $\langle a_1, a_1, a_1, a_1, a_2 \rangle$.

C. Organização do arcabouço

O arcabouço para extensibilidade em linguagens de programação atua como uma camada sobre a linguagem de programação a ser estendida, permitindo que mudanças estruturais definidas pelo usuário sejam incorporadas à linguagem base, de forma modular e transparente. Uma possível organização do arcabouço proposto é apresentada na Figura 10.

De acordo com a Figura 10, uma possível organização do arcabouço proposto inclui os seguintes componentes principais:

– *Mecanismo de tratamento de extensões:* introduzido na Subseção III-B, este componente atua na manipulação

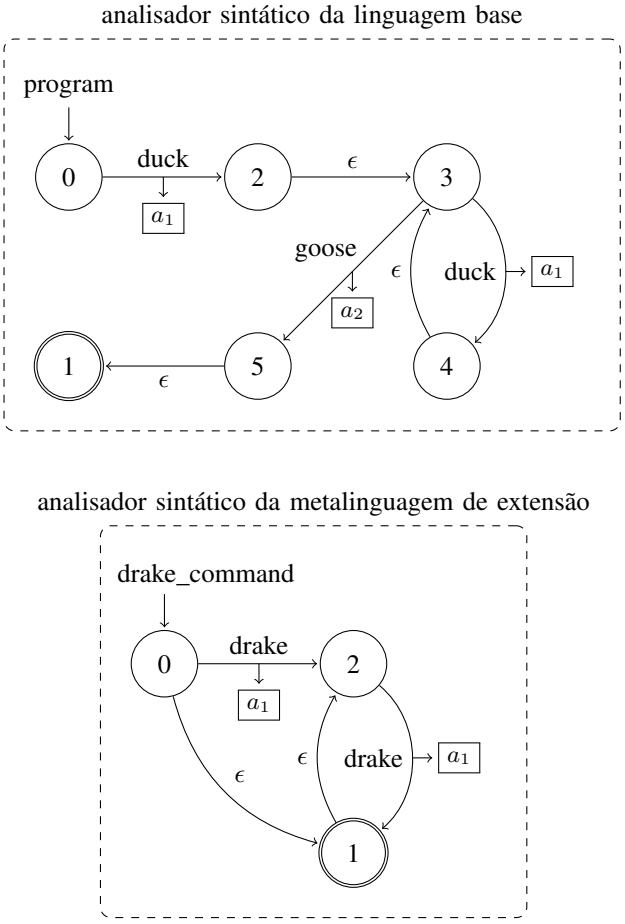

Figura 8. Analisadores sintáticos da linguagem base definida através da gramática da Listagem 2 e da especificação da extensão `drake` a partir da metalinguagem apresentada na Listagem 3.

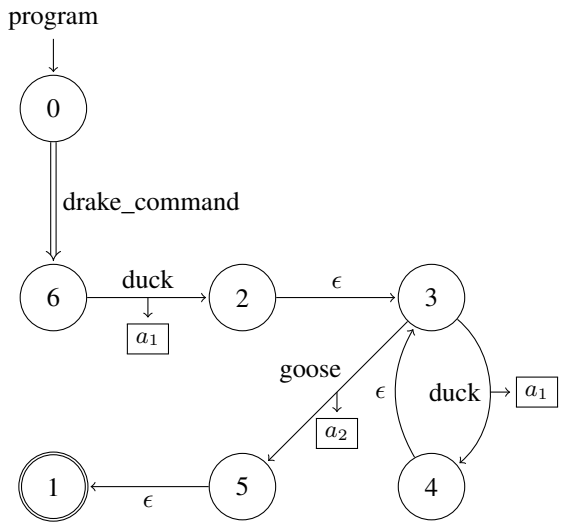

Figura 9. Analisador sintático alterado pelo mecanismo de tratamento de extensões, através de ações adaptativas, para acomodar a nova situação de ocorrência da palavra `drake` na sentença.

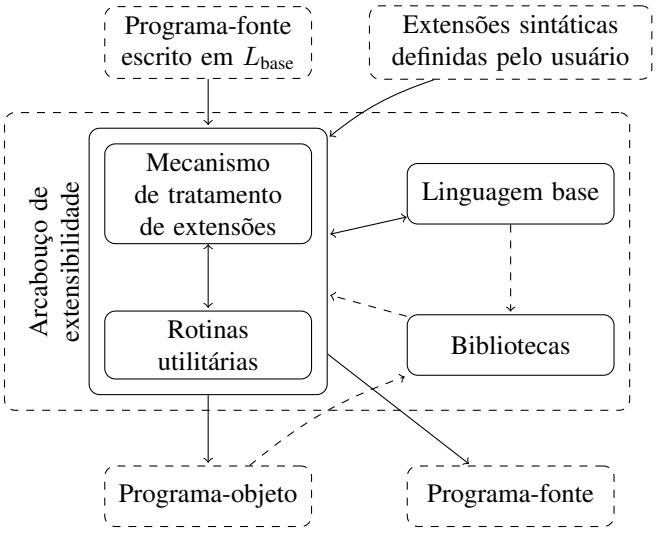

Figura 10. Uma possível organização do arcabouço proposto.

de mudanças estruturais definidas pelo usuário sobre a linguagem base. O mecanismo de tratamento de extensões analisa a especificação das extensões sintáticas escritas em uma metalinguagem de extensão e monitora a análise sintática do programa-fonte escrito na linguagem base, procurando por padrões que coincidam com as regras definidas pelo usuário.

– *Rotinas utilitárias:* o arcabouço apresenta um conjunto de rotinas utilitárias que auxiliam o mecanismo de extensão durante o processo de reconhecimento e análise da linguagem base e das extensões definidas. Em geral, tais rotinas incluem a manipulação de estruturas de dados e eventuais chamadas a bibliotecas do sistema operacional.
– *Acoplamento da linguagem base:* a característica modular do arcabouço proposto permite o acoplamento da linguagem base no contexto do processo de extensão. Deseja-se que a linguagem base possua mecanismos de comunicação para realizar a troca de informações entre o mecanismo de tratamento de extensão, as rotinas utilitárias e o próprio compilador da linguagem base.
– *Bibliotecas:* as extensões sintáticas definidas podem utilizar chamadas de bibliotecas do sistema operacional ou em nível do usuário. Além disso, é possível utilizar um conjunto de bibliotecas pré-definidas para estender a linguagem base, como por exemplo, a adição de construtos adaptativos definidos em biblioteca de propósito geral.

O arcabouço proposto pode resultar, como saída de seu processamento, em um programa-objeto a ser executado pelo sistema operacional (opcionalmente, utilizando chamadas de bibliotecas) ou em um programa-fonte que transforma as extensões sintáticas em comandos válidos da linguagem base, permitindo sua posterior geração de código por um compilador tradicional da linguagem base.

IV. Discussões

O arcabouço para extensibilidade proposto neste artigo almeja proporcionar uma estrutura de suporte para linguagens de programação, permitindo a extensão das mesmas através de modificações sintáticas em tempo de compilação. Tais modificações podem contribuir para o aumento do poder expressivo da linguagem em questão (por exemplo, em [8], o autor introduz o comando `while` em uma linguagem contendo apenas `if` e `goto`, o que aumenta consideravelmente a expressividade da linguagem).

É importante notar que o acoplamento da linguagem base depende de um protocolo de comunicação com o arcabouço; em outras palavras, é imperativo que a linguagem base ofereça mecanismos para troca de informações durante a análise sintática e geração de código. Caso o compilador ou interpretador da linguagem base não ofereça tais subsídios, é necessário intervir no código-fonte da ferramenta e adicionar manualmente os pontos de comunicação; como último recurso, caso o código-fonte não esteja disponível, a extensibilidade pode ocorrer na forma de chamadas a bibliotecas específicas que implementem as funcionalidades definidas através das extensões.

A possibilidade de extensão de linguagens de programação, através do arcabouço proposto neste artigo, pode proporcionar o projeto e implementação de construtos especiais para a escrita de programas com características adaptativas. A adição de uma camada adaptativa em linguagens de programação tradicionais por meio de extensibilidade pode conferir um novo modelo para desenvolvimento de software auto-modificável sem a necessidade imediata de mudanças significativas na forma de escrita de código de um conhecimento profundo, por parte do usuário, dos conceitos de adaptatividade.

V. Considerações finais

Este artigo apresentou um arcabouço que ofereça extensibilidade em linguagens de programação, permitindo a adição ou a alteração de construtos sintáticos definidos pelo usuário através de uma metalinguagem de extensão; o mecanismo de tratamento de extensão monitora a análise da linguagem base, realizando alterações de acordo com as extensões sintáticas definidas. Como saída, o arcabouço permite a obtenção de um programa-objeto a ser executado pelo sistema operacional ou de um programa-fonte que transforma as extensões sintáticas em comandos válidos da linguagem base.

A extensibilidade é um poderoso recurso para a adição e alteração dos construtos sintáticos em linguagens de programação, aumentando seu poder expressivo; espera-se que, através do arcabouço proposto neste artigo, o processo de extensão de uma linguagem de programação seja simplificado de tal forma que o usuário possa beneficiar-se das novas notações, operações ou estruturas de controle para a escrita de programas mais consistentes. Além disso, construtos especiais podem viabilizar o acréscimo de uma camada adaptativa em linguagens de programação tradicionais, permitindo a escrita de programas com características adaptativas.

Referências

[1] J. José Neto, "Contribuições à metodologia de construção de compiladores," Tese de livre docência, Escola Politécnica da Universidade de São Paulo, São Paulo, 1993.
[2] ——, "Adaptive automata for context-sensitive languages," *SIGPLAN Notices*, vol. 29, no. 9, pp. 115–124, set 1994.

[3] J. José Neto, “Um levantamento da evolução da adaptatividade e da tecnologia adaptativa,” *IEEE Latin America Transactions*, vol. 5, pp. 496–505, 2007.
[4] H. Pistori, “Tecnologia em engenharia de computação: estado da arte e aplicações,” Tese de Doutorado, Escola Politécnica, Universidade de São Paulo, 2003.
[5] R. L. A. Rocha and J. José Neto, “Uma proposta de linguagem de programação funcional com características adaptativas,” in *IX Congreso Argentino de Ciencias de la Computación*, 2003.
[6] A. V. Freitas and J. José Neto, “Adaptive languages and a new programming style,” in *WSEAS International Conference on Applied Computer Science*, Tenerife, Canary Islands, Spain, 2006.
[7] A. V. Freitas, “Considerações sobre o desenvolvimento de linguagens adaptativas de programação,” Tese de Doutorado, Escola Politécnica, Universidade de São Paulo, 2008.
[8] A. A. Castro Junior, “Aspectos de projeto e implementação de linguagens para codificação de programas adaptativos,” Tese de Doutorado, Escola Politécnica, Universidade de São Paulo, 2009.
[9] E. J. Pelegrini, “Códigos adaptativos e linguagem de programação adaptativa: conceitos e tecnologias,” Dissertação de Mestrado, Escola Politécnica, Universidade de São Paulo, 2009.
[10] S. R. B. Silva, “Software adaptativo: método de projeto, representação gráfica e implementação de linguagem de programação,” Dissertação de Mestrado, Escola Politécnica, Universidade de São Paulo, 2011.
[11] D. Zingaro, “Modern extensible languages,” Dissertação de Mestrado, McMaster University, Hamilton, Ontario, Canadá, 2007.
[12] J. José Neto, “Adaptive rule-driven devices: general formulation and case study,” in *International Conference on Implementation and Application of Automata*, 2001.
[13] N. Wirth, “What can we do about the unnecessary diversity of notation for syntactic definitions?” *Commun. ACM*, vol. 10, no. 11, pp. 822–823, nov 1977.
[14] K. Cooper and L. Torczon, *Construindo Compiladores*, 2nd ed. Elsevier, 2014.
[15] J. José Neto, *Introdução à compilação*, ser. Engenharia de Computação. LTC – Livros Técnicos e Científicos, 1987.

Paulo Roberto Massa Cereda é graduado em Ciência da Computação pelo Centro Universitário Central Paulista (2005) e mestre em Ciência da Computação pela Universidade Federal de São Carlos (2008). Atualmente, é doutorando do Programa de Pós-Graduação em Engenharia de Computação do Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo, atuando como aluno pesquisador no Laboratório de Linguagens e Técnicas Adaptativas do PCS. Tem experiência na área de Ciência da Computação, com ênfase em Teoria da Computação, atuando principalmente nos seguintes temas: tecnologia adaptativa, autômatos adaptativos, dispositivos adaptativos, linguagens de programação e construção de compiladores.

João José Neto é graduado em Engenharia de Eletricidade (1971), mestre em Engenharia Elétrica (1975), doutor em Engenharia Elétrica (1980) e livre-docente (1993) pela Escola Politécnica da Universidade de S ão Paulo. Atualmente, é professor associado da Escola Politécnica da Universidade de São Paulo e coordena o LTA – Laboratório de Linguagens e Técnicas Adaptativas do PCS – Departamento de Engenharia de Computação e Sistemas Digitais da EPUSP. Tem experiência na área de Ciência da Computação, com ênfase nos Fundamentos da Engenharia da Computação, atuando principalmente nos seguintes temas: dispositivos adaptativos, tecnologia adaptativa, autômatos adaptativos, e em suas aplicações à Engenharia de Computação, particularmente em sistemas de tomada de decisão adaptativa, análise e processamento de linguagens naturais, construção de compiladores, robótica, ensino assistido por computador, modelagem de sistemas inteligentes, processos de aprendizagem automática e inferências baseadas em tecnologia adaptativa.