RONALDO LIMA ROCHA CAMPOS

INTRODUÇÃO DE AÇÕES SEMÂNTICAS NA FERRAMENTA GALS

Florianópolis

dezembro de 2009

RONALDO LIMA ROCHA CAMPOS

INTRODUÇÃO DE AÇÕES SEMÂNTICAS NA FERRAMENTA GALS

Monografia apresentada na Universidade Federal de Santa Catarina para obtenção do título em Bacharel em Ciências da Computação.

Orientador:

Prof$^{\underline{o}}$ Dr. Ricardo Azambuja Silveira

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE BACHARELADO EM CIÊNCIAS DA COMPUTAÇÃO

Florianópolis

dezembro de 2009

Trabalho de conclusão de curso sob o título *Introdução de Ações Semânticas na Ferramenta GALS* defendido por Ronaldo Lima Rocha Campos e aprovada em 03 de novembro de 2009 Florianópolis, Estado de Santa Catarina, pela banca examinadora constituída pelos professores:

Prof$^{\underline{o}}$ Dr. Olinto José Varela Furtado
Universidade Federal de Santa Catarina - UFSC

Prof$^{\underline{o}}$ Dr. José Eduardo De Lucca
Universidade Federal de Santa Catarina - UFSC

Orientador:

Prof$^{\underline{o}}$ Dr. Ricardo Azambuja Silveira
Universidade Federal de Santa Catarina - UFSC

RESUMO

Ao se projetar uma nova linguagem, geralmente recorre-se a ferramentas que auxilie e automatizem esse processo. O GALS é uma ferramente elaborada sobre esses requisitos, auxiliar e automatizar o processo de projetar uma linguagem. Porém, nessa ferramenta, a realização da descrição semântica e sua implementação são tarefas difíceis e trabalhosas. Nesse trabalho, desenvolveu-se extensões ao GALS, de forma a dar melhor suporte a análise semântica. Desenvolveu-se o método de gramática de atributos e um melhor suporte à técnica de tradução dirigida pela sintaxe.

Palavras-chave: Ferramenta GALS, Métodos de Análise Semântica, Gramática de Atributos, Tradução Dirigida por Sintaxe.

ABSTRACT

When designing a new language, usually use tools to assist and automate this process. GALS is a tool developed about these requirements, helping and automating the process of designing a language. However, in this tool the description and implementation of semantics issues are difficult and is a laborious task. In this work, developed extensions to GALS, in order to do a better support of semantic analysis issues. We developed a method of attribute grammar and a better support to technique of syntax driver translation.

Key Words: GALS Tool, Methods of Semantic Analysis, Attributes Grammar, Syntax Driver Translation.

LISTA DE FIGURAS

Figura 1 *Modelo fase de análise de um compilador* 12

Figura 2 *Funcionamento do Lex/Yacc* 17

Figura 3 *Parser Yacc* 18

Figura 4 *Tabela de Símbolos Encadeadas* 33

SUMÁRIO

RESUMO

ABSTRACT

1 INTRODUÇÃO p. 5

1.1 MOTIVAÇÃO p. 5

1.2 OBJETIVOS GERAIS p. 6

1.3 OBJETIVOS ESPECÍFICOS p. 6

2 REVISÃO BIBLIOGRÁFICA p. 8

2.1 TEORIA DAS LINGUAGENS FORMAIS p. 8

2.1.1 LINGUAGENS REGULARES p. 8

2.1.2 AUTÔMATOS FINITOS p. 9

2.1.3 LINGUAGENS LIVRES DO CONTEXTO p. 10

2.2 COMPILADORES p. 11

2.2.1 ANÁLISE LÉXICA p. 12

2.2.2 ANÁLISE SINTÁTICA p. 14

2.2.3 ANÁLISE SEMÂNTICA p. 14

2.2.4 GRAMÁTICA DE ATRIBUTOS p. 15

2.3 GERADORES DE COMPILADORES p. 16

2.3.1 GALS p. 16

2.3.2 LEX/YACC p. 17

2.3.3 ANTLR p. 18

3 GALS p. 20

3.1 GALS . . . p. 20
3.1.1 CÓDIGO GERADO . . . p. 20
3.1.2 ANALISE SEMÂNTICA . . . p. 21

4 GALS ESTENDIDO . . . p. 22
4.1 GALS ESTENDIDO . . . p. 22
4.1.1 ESPECIFICAÇÃO DA GRAMÁTICA . . . p. 25
4.1.2 TRADUÇÃO DIRIGIDA POR SINTAXE . . . p. 26
4.1.3 GRAMÁTICA DE ATRIBUTOS . . . p. 27
4.1.4 CÓDIGO FONTE GERADO . . . p. 30
4.1.5 TESTES REALIZADOS . . . p. 32
4.2 CONCLUSÕES . . . p. 37
4.2.1 CONSIDERAÇÕES FINAIS . . . p. 37
4.2.2 TRABALHOS FUTUROS . . . p. 37

REFERÊNCIAS . . . p. 38

1 INTRODUÇÃO

Reconhecidamente, compiladores são de extrema importância para ciências da computação e a toda informática. O mundo como conhecemos, depende das linguagens de programação, pois todo software existente foi escrito em alguma linguagem de programação. Também todo o hardware que executa algum software, é regido por um tipo de linguagem (linguagem objeto). Porém, antes que um programa (software) possa ser executado, ele primeiro precisa ser traduzido de alguma linguagem de programação, para a linguagem objeto. Esta tarefa é realizada por um compilador.

O presente trabalho de conclusão de curso propõe estender a ferramenta GALS (GESSER, 2003). Tal ferramenta é um gerador de compiladores, ou seja, é uma ferramenta para geração automática de analisadores léxicos, sintáticos, e semânticos, fases do processo de compilação. Seus resultados atendem bem suas expectativas, e a ferramenta é muito utilizada. Porém, conforme foi sendo realizado o estudo de sua implementação, constatou-se uma necessidade de estender suas funcionalidades, para prover melhores técnicas de suporte a implementação de ações semânticas, principal objetivo deste trabalho.

1.1 MOTIVAÇÃO

O projeto de um compilador é o casamento entre teoria e pratica. É uma área com imensa fundamentação teórica, que de fato é usado, na pratica, em seus complexos algoritmos.

Até alguns anos atrás, haviam poucas linguagens de programação, e poucas arquiteturas de hardware. Hoje, há centenas de linguagens, e diversas arquiteturas de hardware. Os compiladores tiveram também que acompanhar essa evolução.

Ao se comparar compiladores antigos com modernos, vemos grandes mudanças. Compiladores antigos, eram geralmente implementados por uma única pessoa, ou um pequeno grupo, possuíam algumas centenas de linhas de código, e eram feitos em linguagens de baixo nível. Hoje, um projeto de compilador envolve equipes inteiras, recursos avançados

de engenharia de software, fazendo uso de componentes já existentes e *frameworks* de auxilio a construção de compiladores. Possuindo, assim, milhares de linhas código.

A construção de um compilador completo é muito complexa, demanda tempo, recursos e conhecimento, sendo um trabalho oneroso as vezes desnecessário, pois, etapas como análise léxica e sintática podem ser automatizadas por geradores de compiladores.

Dessa forma o desenvolvimento de um compilador pode ser melhor focado nas etapas de descrição da linguagem, pois, desta forma, o projetista não precisa tomar conhecimento dos diversos algoritmos envolvidos nessas etapas, análise léxica e sintática.

Hoje exitem diversas ferramentas que propõem o mesmo objetivo, tais como, as consagradas Lex (SCHMIDT; LESK,) e Yacc (JOHNSON, 1974), Antlr (PARR, 2007), dentre outras. Porém, elas apresentam alguns problemas: algumas fazem somente a análise léxico-sintática ou geração de código, o que obriga uso de mais de uma ferramenta, levando, muitas vezes, à incompatibilidade no uso dos analisadores gerados. Ou usam apenas uma das diversas técnicas de análise para a construção do *parser*.

Outro fator importante é a falta de documentação das soluções existentes que, em muitos casos são restritas somente ao código fonte, e a alguns exemplos. Uma ferramenta simples, porém que alie tecnologias e facilidades existentes, faz-se necessário.

1.2 OBJETIVOS GERAIS

O presente trabalho tem como objetivo desenvolver extensões na ferramenta GALS, que auxilie o desenvolvimento da etapa da análise semântica de compiladores e interpretadores de linguagens de programação, fazendo uso de uma linguagem de descrição da gramática e suas regras. A ferramenta deve, de forma automática, gerar as principais partes de um compilador: o analisador léxico e o sintático, bem como prover técnicas de suporte a implementação de ações semânticas.

1.3 OBJETIVOS ESPECÍFICOS

1. Compreender como funciona e como é implementado um compilador;
2. Analisar as diferentes técnicas e algoritmos de parser;
3. Estudar e analisar a ferramenta GALS, para identificar dificuldade e identificar os aspectos que pode ser aperfeiçoados;

4. Desenvolver, na ferramenta, as extensões possíveis e necessárias para dar apoio a fase de analise; semântica.

2 *REVISÃO BIBLIOGRÁFICA*

2.1 TEORIA DAS LINGUAGENS FORMAIS

Teoria das linguagens foi desenvolvida na década de 1950, com o objetivo inicial de desenvolver teorias relacionadas com as linguagens naturais. No entanto, sua importância foi verificada em diversas outras áreas, em especial no estudo de linguagens artificiais e linguagens originárias da Computação e Informática (MENEZES., 2000).

Em computação uma linguagem formal pode ser formalmente definida como sendo um conjunto finito de símbolos sobre um alfabeto, ou seja, se L é uma linguagem e V um alfabeto, então: L $\subseteq V_\epsilon$, onde V_ϵ* é V $\cup$ $\{\epsilon\}$.

2.1.1 LINGUAGENS REGULARES

Linguagens Regulares são linguagens formais, empregadas na construção de reconhecedores de linguagens. Apesar de fazerem parte de um universo restrito de linguagens, são empregados na construção, particularmente no aspectos léxicos, de linguagens de computação. Formalmente, se L é uma linguagem regular, então ela pode ser reconhecida, ou descrita, através de autômatos finitos, expressões regulares ou gramáticas regulares.

As linguagens regulares podem ser definidas através de uma gramática regular. Está por sua vez pode ser formalizada como segue: G = (V_n, V_t, P, S), onde:

1. V_n é um conjunto de símbolos, denominados não-terminais, são utilizados na descrição da linguagem;

2. V_t é um conjunto de símbolos, denominados terminais, são os símbolos de fato utilizados na formação das sentenças da linguagem;

3. P é um conjunto finito de produções, na forma P(α, β) onde $\alpha \in V_n$, deriva $\beta \in$ ($V_n \cup V_t$).

4. S é o símbolo inicial da gramática.

2.1.2 AUTÔMATOS FINITOS

Em Teoria da Computação, autômatos finitos são modelos computacionais, simples o bastante para se construir uma teoria matemática manejável sobre eles. Autômatos são uma especie de computador abstrato, fundados em dois princípios simples, porém muito poderosos, memória (estados) e transições. Dassa forma consegue-se representar uma série de problemas complexos.

AUTÔMATOS FINITOS DETERMINÍSTICOS

Autômatos finitos determinísticos (do inglês *deterministic finite automata*, DFA) são o tipo mais simples de autômatos. O funcionamento de um DFA assemelha-se ao de uma máquina, composta por três partes, um registro, ou fita, uma unidade de controle, e um programa de transição. A partir da fita de entrada, a unidade de controle a lê, um simbolo por vez. E conforme o simbolo lido, o programa de transição determina um novo estado à máquina. Ao final da leitura da fita, se a máquina estiver em um estado especial, denominado aceitação, a fita é reconhecida, caso contrário, rejeitada (SIPSER., 2007).

Um automato finito é formalmente definido como sendo uma 5-upla (Q, Σ, δ, q_0, F), onde

1. Q é um conjunto finito de estados;
2. Σ é um conjunto de símbolos, chamados alfabeto;
3. δ é uma função de transição do tipo: Q × Σ → Q;
4. q_0 é o estado inicial do automato, onde $q_0 \in Q$;
5. $F \subseteq$ Q é o conjunto de aceitação.

AUTÔMATOS FINITOS NÃO-DETERMINÍSTICOS

Autômatos finitos não-determinísticos (do inglês *nondeterministic finite automata*, NFA) diferentemente de máquinas determinísticas, as quais conhecendo seu estado e o próximo símbolo de entrada, saberemos qual será seu próximo estado, para o não-determinismo, um conjunto de estados é esperado, ou seja, há vários "caminhos" que podem ser seguidos de forma paralela.

Um automato finito não-determinísticos é formalmente definido como sendo uma 5-upla (Q, Σ, δ, q_0, F), onde

1. Q é um conjunto finito de estados;
2. Σ é um conjunto de símbolos, chamados alfabeto;
3. δ é uma função de transição do tipo: Q $\times$ $\Sigma_\epsilon \rightarrow$ P(Q);
4. q_0 é o estado inicial do automato, onde $q_0 \in Q$;
5. $F \subseteq$ Q é o conjunto de aceitação;
6. ϵ é a palavra vazia.

Sendo, Σ_ϵ definido como: $\Sigma \cup \{\epsilon\}$, e P(Q) é o conjunto das partes de Q, ou seja, para qualquer conjunto Q temos P(Q) como sendo a coleção de todos os subconjuntos de Q.

EXPRESSÕES REGULARES

Trata-se de um gerador, pois toda linguagem regular pode ser descrita por uma expressão simples, pode-se então, inferir como construir tal linguagem. Uma expressão regular é definida a partir de conjuntos básicos de símbolos e operações sobre eles, união e concatenação e fechamento. São adequadas para descrever linguagens, pois dada sua forma, expressão matemática, é bem pertinente a comunicação.

Segue sua definição formal, R é uma expressão formal se:

1. Qualquer símbolo α pertencente a algum alfabeto Σ;
2. ϵ, que representa a linguagem contendo exclusivamente a palavra vazia;
3. $(R_1 \cup R_2)$ onde R_1 e R_2 são expressões regulares;
4. $(R_1 \circ R_2)$ onde R_1 e R_2 são expressões regulares;
5. (R_1^*), onde R_1 é uma expressão regular.

2.1.3 LINGUAGENS LIVRES DO CONTEXTO

A classe de linguagens livre do contexto compreende um universo mais amplo e possuem como um de seus subconjuntos as linguagens regulares. Seu estudo é de extrema importância para a informática, pois por serem mais representativas, lidam com uma classe de problemas comum às linguagens de programação, como por exemplo, balanceamento de expressões, construções de blocos, entre outras (SIPSER., 2007).

Seu estudo é desenvolvido através de um formalismo axiomático, ou seja, uma gramática, e um reconhecedor.

GRAMÁTICAS LIVRE DE CONTEXTO

Foram usadas primeiramente no estudo de linguagens humanas. Constituem-se em um método mais poderoso de descrever linguagens, descrevem características que possuem estrutura recursiva (MENEZES., 2000).

Sua principal aplicação ocorre na compilação de linguagens de programação. Em sua maioria, as linguagens de programação são são definidas formalmente a partir de uma gramática estendida, denominada EBNF.

Sua definição formal é dada como, G = (V_n, V_t, P, S), onde:

1. V_n é um conjunto de símbolos, denominados não-terminais, são utilizados na descrição da linguagem;

2. V_t é um conjunto de símbolos, denominados terminais, são os símbolos de fato utilizados na formação das sentenças da linguagem;

3. P é um conjunto finito de produções, na forma P(α, β) onde $\alpha \in V_n$, deriva $\beta \in (V_n \cup V_t)^*$.

4. S é o símbolo inicial da gramática.

Sendo que P é da forma $\alpha \rightarrow \beta$, onde α é uma variável de V_n e β é uma palavra de $(V_n \cup V_t)$.

2.2 COMPILADORES

Compiladores são programas que tratam de linguagens, manipulando-as de diferentes formas. Basicamente transformam sentenças escrita em uma linguagem fonte em outra em linguagem alvo, ou seja, a partir de sentenças em uma linguagem de entrada, é gerada uma linguagem de saída, e ambas possuem o mesmo significado e comportamento. Esse processo é denominado de tradução.

De forma bem simples um compilador pode ser definido como um programa que, como entrada recebe um programa fonte (em linguagem de programação), e o traduz para um

programa equivalente em outra linguagem, em sua maioria linguagem objeto, capaz de ser executado por um computador alvo (LOUDEN, 2004).

Porém, dependendo de sua saída, o compilador pode ainda ser classificado como (AHO et al., 2008, p. 1-2):

- Pré-processador, se a linguagem de entrada for a mesma de saída;
- Interpretador, se a partir de uma linguagem de entrada, forem geradas diretamente ações em um computador.

Na maioria dos casos o processo completo de compilação pode ser dividido em duas partes, análise e síntese.

Na análise, o compilador subdivide o programa fonte em partes e é gerado uma estrutura gramatical sobre elas. A partir dessa estrutura é criado uma representação intermediária do programa fonte. Nessa parte a análise verifica o programa sintática e semanticamente, e coleta informações sobre dados e comportamento do programa para passar à próxima fase. Caso o programa fonte esteja incorreto, é desejável que o compilador gere uma análise dos erros (AHO et al., 2008).

A parte referente a síntese irá então construir o programa objeto, usando essa representação intermediária e as informações retiradas, que muitas vezes recebe o nome de tabela de símbolos.

Por sua vez o processo de análise pode ser subdividido nas seguintes partes: análise léxica, análise sintática e análise semântica. A figura 1 mostra o processo de análise.

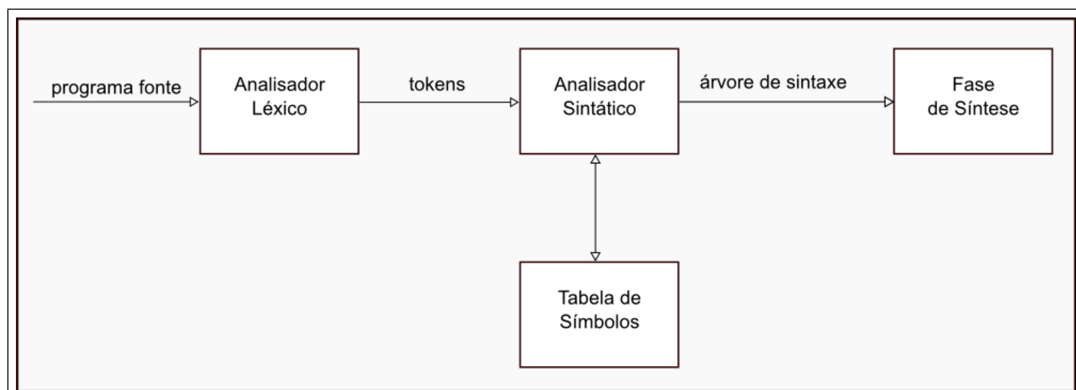

Figura 1: *Modelo fase de análise de um compilador*

2.2.1 ANÁLISE LÉXICA

Para a discussão sobre análise léxica, precisamos definir alguns termos importantes:

- *Token* é um símbolo abstrato, que representa uma unidade léxica, cada *token* representa um padrão de caracteres ou identificadores. Os *tokens* são os símbolos de entrada que o analisador sintático processa.
- Lexema são seqüências de caracteres que casam com o padrão de um *token*, e é identificado como uma instância deles.

A Análise Léxica corresponde a primeira fase do processo de compilação. Sua principal tarefa é ler os caracteres da entrada do programa, agrupá-los em lexemas e produzir como saída uma seqüência de *tokens*. Esse fluxo de *tokens* é então enviado ao analisador sintático. Além disso o analisador léxico pode também interagir com a tabela de símbolos.

Normalmente a etapa de analise léxica está integrada a sintática. O analisador sintático faz chamadas ao léxico sempre que precisar reconhecer um novo *token*. Embora fosse possível deixar ao analisador sintático a tarefa de reconhecer os *tokens*, essa etapa é separada, pois:

- Dessa forma o projeto é simplificado, podendo tratar-se com facilidade detalhes da especificação da linguagem e formatação dos caracteres, como quebras de linha, e outras.
- Sua eficiência é melhorada, uma vez que podemos aplicar técnicas especializadas relacionadas apenas à tarefa léxica.
- A portabilidade é melhorada, mudanças nos dispositivos de entrada do compilador podem ser mais facilmente tratados.

Para transformar o fluxo de caracteres de entrada em uma seqüência de *tokens*, reconhecendo dessa forma, palavras, identificadores e constantes, o analisador léxico é implementado como um reconhecedor de linguagens regulares. Dessa forma sua implementação faz uso, a partir de uma especificação, de autômatos finitos.

Essa forma garante eficiência e simplicidade. Porém é comum precisarmos ler alguns caracteres adiante, afim de descobrir sobre qual *token* casamos nosso padrão. Podemos resolver esse problema com uma técnica bem simples, chamada *lookahead*, basta, ao se verificar que um lexema pode ser aplicado a mais do um *token*, que se mantenha o estado atual da analise, e se caminhe para o próximo símbolo, afim de se obter informações suficientes para casar apenas um padrão.

A tarefa de construir um analisador léxico pode ser inteiramente automatizada, bastando apenas uma especificação formal da linguem a ser reconhecida e um software gerador de analisadores léxicos.

2.2.2 ANÁLISE SINTÁTICA

A análise sintática é o processo pelo qual determina-se a sintaxe, ou estrutura, de um programa. Ou seja, determina se uma cadeia de símbolos terminais pode ser gerada por uma gramática. A análise sintática está associada às gramáticas livre do contexto, dessa forma é conveniente expressar suas derivações através de uma árvore, que chamamos de árvore sintática.

A maioria dos algoritmos e métodos de análise sintática estão divididos em duas classes, descendentes e ascendentes. Essas classes referem-se à ordem com que os nós das árvores são construídos.

ANÁLISE ASCENDENTE

A analise ascendente corresponde à construção da árvore de derivação, para sua cadeia de entrada, de baixo para cima, ou seja a partir de suas folhas, rumo ao topo da árvore. Essa classe de analisadores pode tratar de um maior número de gramáticas e esquemas de tradução. Porém na prática, e em seus algoritmos, tal árvore dificilmente é construída. Por tais fatores são utilizados em geradores de analisadores sintáticos.

ANÁLISE DESCENDENTE

O método de análise descendente consiste em construir a árvore de derivação, para a cadeia de entrada, de cima para baixo, ou seja, da raiz da árvore em direção às folhas.

Sua popularidade dá-se ao fato de que analisadores podem ser construídos de forma mais fácil dessa forma.

2.2.3 ANÁLISE SEMÂNTICA

Como estrutura à análise semântica temos algumas técnicas bem formuladas, como a geração de tabelas de símbolos, grafos de dependência e árvores de derivação.

Como suporte, uma técnica muito utilizada e de simples implementação é a tradução dirigida por sintaxe.

TABELA DE SÍMBOLOS

São estruturas de dados que contem informações sobre as construções do programa fonte. Essas informações são obtidas pelas fase de análise, e usadas na análise semântica e na fase de síntese.

Tal tabela contém informações sobre os identificadores como, seu nome, lexema, seu tipo, endereço em memória, qualquer informação que julgue-se importante.

GRAFO DE DEPENDÊNCIAS

Os grafos de dependências representam o fluxo de informações entre instâncias dos atributos em uma árvore de derivação. Seus nós são os atributos de uma produção específica e suas arestas expressão as regras semânticas entre os nós.

TRADUÇÃO DIRIGIDA POR SINTAXE

Esse tipo de técnica associa informações, atributos, aos símbolos da gramática, de forma a, durante a análise das produções e derivações, aplicar as regras semânticas ao programa fonte.

2.2.4 GRAMÁTICA DE ATRIBUTOS

Desenvolvida por Donald E. Knuth, a gramática de atributos (KNUTH, 1968) é um dos métodos mais simples para se atribuir significados às sentenças de uma linguagem. Método operacional, usado para vários fins, como os citados em (LANG, 2003):

- Aumentar o poder de reconhecimento de uma gramática livre de contexto ou regular;
- Mapeamento direto entre construções sintáticas e instruções, para geração de código;
- Verificação de regras estáticas da semântica, tais como verificação escopo, tipos de variáveis, inferência de tipos de variáveis;
- Especificação semântica de construções sintáticas.

Tal técnica consiste-se em definir atributos para os símbolos da gramática e suas produções. Dessa forma pode-se definir valores, ou acessa-los, aos símbolos da gramática, contidos na própria derivação apenas.

Pode ser representado junto a árvore de derivação. E suas regras são processadas quando o *parser* é executado para processar uma sentença da linguagem, trabalhando em conjunto com o analisador sintático.

Quanto a sua implementação, são dependentes do tipo do *parser* usado, pois seus atributos podem ser herdados se o *parser* for descendente, ou sintetizados se for ascendente (AHO et al., 2008).

2.3 GERADORES DE COMPILADORES

Como suporte ao desenvolvimento de um compilador, algumas ferramentas foram desenvolvidas. Elas vão desde um simples gerador de reconhecedores de linguagens regulares, até complexas ferramentas de geração e otimização de código objeto.

Algumas das mais importantes serão tratadas a seguir.

2.3.1 GALS

O GALS é uma ferramente de geração de analisadores léxicos e sintáticos. Seu uso é feito através de sua interface gráfica, onde são definidos os *tokens* e produções da gramática. Ele permite uma série de opções, como gerar somente o analisador léxico, ou somente o sintático, ou ambos de forma integrada, e quanto a análise semântica, é possível utilizar a técnica de tradução dirigida pela sintaxe (GESSER, 2003).

Como parser, também permite a escolha de seu algoritmo, podendo ser:

- Descendentes:

  Descendente Recursivo

  LL.

- Ascendente:

  SRL

  LALR

  LR.

Como, além de auxiliar na construção de compiladores, o GALS se propõe a ser uma ferramenta com auxilio didático, oferece uma série de opções para esse fim, como a visualização das tabelas de analise léxica e sintática, e os conjuntos *First* e *Follow*.

Tem sido muito utilizado no ensino de compiladores. Pois, possui também um simulador, permitindo testar o funcionamento da gramática. Com o GALS é possível gerar o compilador em diversas linguagens, C++, Java e Pascal.

FUNCIONAMENTO

Seu funcionamento é semelhante as demais ferramentas. Uma linguagem de descrição da gramática é utilizada para especificar a linguagem, e a partir dai é possível gerar os analisadores léxico e sintático, em uma das suas opções, já citadas.

O GALS fornece também um simulador léxico, e um sintático. O funcionamento do simulador léxico é bem simples, insere-se uma sentença e ele mostra os *tekens* identificados. Já o simulador sintático, ao se inserir uma sentença, mostra sua árvore de derivação.

2.3.2 LEX/YACC

O Lex foi projetado para o processamento léxico de caracteres. Ele aceita uma especificação que reconheça um conjunto de caracteres, ou seja, uma linguagem regular. E produz um programa em linguagem de propósito geral, nesse caso, em linguagem C, capaz de reconhecer tal especificação (SCHMIDT; LESK,).

O funcionamento do Lex dá-se da seguinte forma, como entrada, o compilador Lex recebe uma especificação da linguagem a ser reconhecida, e produz, como saída, um programa em linguagem C, que reconhece tal linguagem. Como exemplo a figura a seguir (SCHMIDT; LESK,):

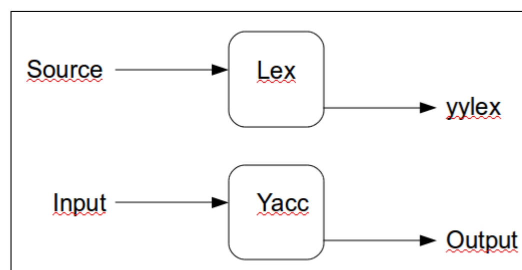

Figura 2: *Funcionamento do Lex/Yacc*

Normalmente, por apenas reconhecer linguagens regulares, o Lex é usado para gerar um programa, que é utilizado como rotina em um analisador sintático (AHO et al., 2008, p. 90). Sendo muito utilizado em conjunto com o Yacc.

Por sua vez o Yacc, diferente do Lex, foi criado com o propósito de reconhecer linguagens livre de contexto. Reconhecendo desta forma as linguagens de programação

(JOHNSON, 1974).

O Yacc, tem como função gerar um analisador sintático, de acordo com uma especificação de linguagem. Essa especificação traz a estrutura de reconhecimento da linguagem de entrada, juntamente com a função, em código para a linguagem C, para ser processado quando uma estrutura é reconhecida.

Seu funcionamento, semelhante ao Lex, ocorre da seguinte forma, o compilador recebe a especificação da linguagem, e produz, em linguagem C, um parser para ela. Porém, esse parser, como entrada, espera receber já um fluxo de *tokens*, e não um conjunto de caracteres. Por tal fato, o Yacc precisa trabalhar em conjunto com um analisador léxico.

Dessa forma o Yacc é normalmente associado ao uso do Lex. Sua estrutura de funcionamento segue como a figura:

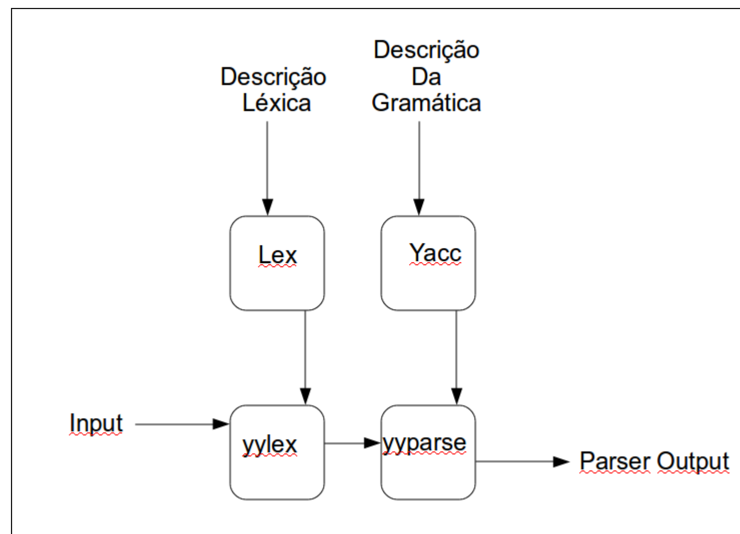

Figura 3: *Parser Yacc*

2.3.3 ANTLR

Atualmente, o Antlr é uma das ferramentas mais utilizadas para o reconhecimento de linguagens. Seu projeto inicial tem mais de vinte anos. E em sua ultima versão, foi completamente reescrito (PARR, 2007).

De forma diferente de outras ferramentas, o Antlr é uma ferramenta completa. Através de uma linguagem de especificação, ele é capaz de gerar um reconhecedor de linguagens, que integra as fazes de analise léxica e sintática.

A ferramenta faz uso de algoritmos que geram um parser LL. Tal escolha deve-se ao fato que parser LL são mais rápidos e requerem menos memória, e, principalmente, são

mais fáceis de implementar mecanismos de detecção e recuperação de erros.

Apesar de ser implementado em linguagem de programação Java, o Antlr pode gerar reconhecedores em diversas linguagens, como C++, Phyton, Java e C#.

Como suporte a análise semântica, faz-se uso do mesmo dispositivo utilizado no Yacc. Ações são descritas juntamente com as produções, e são executadas ao processa-las. Alternativamente, porém, o Antlr oferece um recurso de geração de uma árvore sintática abstrata, onde, dessa forma, as ações semânticas podem ser feitas ao se processar os nós da árvore (PARR., 2007).

3 GALS

3.1 GALS

GALS é um gerador de compiladores, desenvolvido por Carlos Gesser (2003). Tal ferramenta foi concebida como trabalho de conclusão de curso, sob orientação do Prof. Dr. Olinto José Varela Furtado. Sua principal contribuição é o auxilio e automatização das etapas de construção de um compilador. Com destaque para a as fases de analise léxica e sintática.

Tal ferramenta foi produzida em linguagem Java, e sua implementação faz uso de uma interface gráfica, para a definição de sua meta-linguagem para a construção do compilador. O GALS é altamente configurável, sendo possível gerar seu analisador sintático usando vários tipos de técnicas, como “ll”, “slr”, entre outras. Uma melhor descrição pode ser obtida em (GESSER, 2003).

3.1.1 CÓDIGO GERADO

Para utilizar a aplicação GALS para o reconhecimento de linguagens, é preciso definir alguns pontos. Tratando-se de uma aplicação constituída de uma interface gráfica com o usuário, onde há uma divisão para cada estrutura da gramática, é simples sua formulação. Essa estrutura é dividida em um componente para a entrada das definições regulares, um para *Tokens*, um para não terminais e outro para as definições da gramática.

Já as configurações relativas aos módulos gerados, são feitas sobre um menu de opções. Questões relativas ao tipo de entrada do analisador léxico, ou tipo do parser são feitas sobre esse menu. Também são feitas as configurações quanto a linguagem do código gerado.

Ao submeter-se uma gramática e suas regras à ferramenta, ela, após o processo de definições de configurações, gera um conjunto de arquivos fontes, que corresponde a um compilador para a tal gramática.

O conjunto de códigos fonte que formam o compilador, são organizados em tres módulos. São eles:

- Módulo de constantes e tabelas, para o uso nas fazes de análise, ficam agrupados em arquivos diferentes, para facilitar sua checagem;
- Um módulo do núcleo do compilador, formado pelos analisadores, léxico, sintático e semântico;
- Módulo para mensagens de exceções e erros, também organizados em arquivos, conforme sua pertinência, analise léxica, sintática e semântica.

Quanto ao seu funcionamento, o compilador gerado comporta-se seguindo as três fases de análise. Porém, como já consagrado, a fase de análise léxica é codificada como uma rotina do analisador sintático. E também, de forma semelhante, é feita a análise semântica.

3.1.2 ANALISE SEMÂNTICA

No que se trata da análise semântica, um mecanismo muito simples foi desenvolvido. Tal análise é realizada utilizando a técnica de tradução dirigida pela sintaxe.

Sua implementação é bem simples, e consiste em, na especificação da gramática inserir um simbolo "#" seguido de um número, a direita de uma produção. Tal simbolo é identificado, na fase sintática. como sendo um *tokem* de ação semântica. Dessa forma, durante a análise sintática, quando um *tokem* de ação semântica é encontrado, o analisador é parado, e uma rotina do analisador semântico é chamada, sendo informado o código da ação, e uma referência ao ultimo simbolo processado pelo analisador léxico.

Tal modelo, apesar de simples, torná-se muito confuso ao se criar gramáticas mais robustas. Pois conforme a necessidade em se tomar ações, símbolos são inseridos, chegando a um ponto em que tal representação fica carregada de símbolos semânticos e números de controle.

Também não há uma forma clara de se obter mais de uma referência de *tokens* processados pelo fase léxica, o que obriga a criação de estruturas de dados mais robustas, bem como a inserção de regras, que utilizam essas estruturas, para codificar estados e buscar *tokens* já processados, demandando um esforço maior de implementação.

4 GALS ESTENDIDO

4.1 GALS ESTENDIDO

Para melhorar a etapa de análise semântica da ferramenta GALS algumas extensões são propostas. De inicio, é essencial uma melhor abordagem quanto a técnica existente de tradução dirigida pela sintaxe. Ainda assim, uma técnica que permita auxiliar a etapa de analise semântica é muito bem vinda. Com isso, torna-se a ferramente ainda mais robusta e poderosa, quanto ao auxilio à construção de compiladores e interpretadores de linguagens.

Dentre as diversas técnicas de analise semântica, escolheu-se por estender o GALS afim de contemplar duas delas.

São elas, um melhor suporte a tradução dirigida pela sintaxe, a qual o GALS já suportava, porem de forma limitada. E a gramática de atributos.

Escolheu-se por essas extensões, pois tratam-se de técnicas bem conceituadas, que acabam por auxiliar a criação de partes complexas de um compilador. Tanto a tradução dirigida pela sintaxe quanto a gramática de atributos, são técnicas comumente utilizadas na construção de compiladores e interpretadores, dentre suas principais características, tem-se que,

- São de fácil compreensão;
- Seu uso e implementação são simples, e não requerem muitos recursos, e restrições;
- Auxiliam na resolução de problemas comuns a maioria das linguagens de programação, como a verificação de tipo, por exemplo;
- Por serem bem conhecidas, seu uso torna-se simples, e mantem-se a simplicidade em representar a gramática da linguagem.

Porém, para realizar tais mudanças, deve-se tomar alguns cuidados, afim de continuar

garantindo a qualidade e simplicidade da ferramenta. Com base nisso, uma serie de requisitos devem ser tratados.

Como metas deste trabalho na implementação destas extensões pretende-se:

- Tornar a ferramenta mais robusta;
- Prover um melhor suporte a implementação de ações semânticas;
- Manter sua simplicidade de uso;
- Manter sua compatibilidade.

Para manter tais requisitos, optou-se por especializar essas técnicas. Portanto mudanças em sua estrutura básica foram feitas de forma a garantir que os requisitos acima fossem cumpridos.

A técnica de análise semântica que faz uso de gramáticas de atributos está fortemente ligada a criação de uma estrutura em arvore, comumente chamada de árvore sintática. Tal árvore é criada pelo analisador sintático, portanto ela só será concluída ao final dessa etapa de análise.

Tal fato acarreta em mudanças significativas na estrutura interna em que o GALS foi construído. No GALS, optou-se por realizar a tradução dirigida pela sintaxe, pois a implementação do seu gerador de compiladores está fortemente ligada a etapa de análise sintática, conhecida como *parser*. Para se construir a árvore sintática, e a partir dela realizar as ações relativas a gramática de atributos, uma nova estrutura teria que ser construída, tendo-se que modificar, drasticamente, uma parte significativa do GALS. Dessa forma, fere-se os requisitos.

Em virtude desses fatos, restringe-se tal técnica para que ela se adéque aos requisitos. Embora ainda exista a ideia da construção da árvore sintática, essa não é de fato construída, ao menos não em sua totalidade. Ao invés disso, usa-se uma estrutura encadeada de nós, semelhante a grafos, onde cada nó representa um estado, uma produção. De forma que a medida que a análise sintática é feita, estados são criados, e o processamento das ações é realizado em função do desenvolvimento desses estados.

O funcionamento dos estados ocorre de forma simples, neles são armazenados as informações dos atributos da gramática, e sobre esses atributos as ações podem ser tomadas. Embora não exista a árvore sintática, a forma como esses estados se relacionam garante que, de fato, tenha-se todos os dados relativos ao nó da árvore, aqui representado pelo es-

tado em processamento, para a execução das ações. Dessa forma garante-se a linearidade do processo, como ocorre no GALS, e dessa forma, nenhum requisito é quebrado.

Em detalhes, ao se processar a gramática de atributos, uma análise dos atributos necessários à produção corrente é realizada. Tem-se então, para cada produção, um estado e seus atributos. De forma encadeada os estados são criados, a medida em que o parser processa suas regras, de forma que um estado novo é criado sempre que uma nova regra é processada, e tal estado liga-se ao estado anteriormente processado. De forma que os valores dos atributos são obtidos sempre dos nós de suas produções, de forma a comparar com a árvore sintática, os atributos são sempre sintetizados, ou seja são obtidos ao se processar os nós filhos.

Como exemplo há a regra sintática:

```
<E>  ::= <T> <E_>;
```

E sua representação semântica:

```
{<T>.type = <E_>.type}, onde espera-se que o atributo
                        type seja o mesmo entre <T> e <E_>.

{install <T>.type}, onde o atributo type de <E> recebe
                    o valor do mesmo atributo de <T>.
```

Nessa abordagem, o estado "E" é criado com os atributos "type" esperado de "T" e de "E_". Em seguida processa-se a regra "T". Desse processamento, novamente um estado é criado, com uma referência a "E", e ao final de seu processamento, é atribuído a "E" seu valor de atributo. De forma análoga, ocorre o processamento de "E_", e ao final, tem-se o estado "E" com seus atributos devidamente preenchidos. Nesse ponto o processamento da primeira regra é realizado, a comparação dos atributos, e logo depois, o valor do atributo de "T" é atribuído a "E", que por sua vez o propaga, se houver, ao estado acima, o que ele referencia.

Para tanto uma consideração deve ser feita. Ao se trabalhar com tal linearidade, ao invés da construção em árvore, depende-se fortemente do tipo de parser a ser usado, e consequentemente do tipo de gramática a ser desenvolvida. Por detalhes de sua implementação específica, para que tal recurso funcione de forma correta, tem-se que fazer uso de uma parse "ll".

4.1.1 ESPECIFICAÇÃO DA GRAMÁTICA

Como no GALS a especificação da gramática dá-se em três partes, a definição dos símbolos terminais, a definição dos símbolos não-terminais e a definição das produções da gramática.

As extensões propostas alteram a forma como tais produções da gramática são declaradas. As produções são descritas de forma semelhante ao GALS, com algumas extensões. Segue uma forma semelhante a BNF:

```
<NT> ::= <LISTA DE SÍMBOLOS> <AÇÕES SEMÂNTICAS>
       | <LISTA DE SÍMBOLOS> <AÇÕES SEMÂNTICAS> ...;
```

Onde "NT" são símbolos não-terminais, e "LISTA DE SIMBOLOS" são compostos por símbolos terminais ou não, identificadores para as ações semânticas e por algumas ações semânticas. Define-se então da seguinte forma:

```
<LISTA DE SÍMBOLOS> ::=  ID = <SÍMBOLO>
                       | #acao_semantica
                       | <SÍMBOLO>;
```

Onde "SÍMBOLO" representa um símbolo terminal ou não. Já os campos "id" e "#" estão relacionados as ações semânticas descritos em breve.

As "AÇÕES SEMÂNTICAS" são definidas como segue:

```
<AÇÕES SEMÂNTICAS> ::= -> <AÇÕES DE COMPARAÇÃO> <LISTA DE AÇÕES>
                       -> <AÇÕES DE ATRIBUIÇÃO> <LISTA DE AÇÕES> ;
<LISTA DE AÇÕES> ::= , <AÇÕES SEMÂNTICAS>
                    | ϵ ;
```

Onde "->" é o símbolo que marca o inicio das declarações semânticas. "AÇÕES DE COMPARAÇÃO" e "AÇÕES DE ATRIBUIÇÃO" representam os tipos de ações semânticas a serem declarados. Defini-sem como:

```
<AÇÕES DE COMPARAÇÃO> ::=  ID . CAMPO = ID . CAMPO ;

<AÇÕES DE ATRIBUIÇÃO> ::=  install ID . CAMPO
```

```
                        | install lookup ( ID , CAMPO )
                        | install " CAMPO " " VALOR "
                        | install î
```

Os outros símbolos, “ID”, “CAMPO” são representados pela expressão regular:

```
( a-z | A-Z | _ ) . ( a-z | A-Z | _ | 0-9 )*
```

Já “VALOR” aceita qualquer sequência de caracteres.

4.1.2 TRADUÇÃO DIRIGIDA POR SINTAXE

Como suporte a tradução dirigida pela sintaxe, algumas alterações foram feitas, para facilitar a tomada de ações.

Para facilitar a construção das ações semânticas e acabar com a confusão de faze-las utilizando números, tais ações são referenciadas pelo nome da rotina que as tratam. Como exemplo tem-se o fragmento da gramática a seguir:

```
<E>  ::= <T> <E_>;
<E_> ::= "+" <T> #2 <E_>
       | "-" <T> #3 <E_>
       | î;
<T>  ::= <F> <T_>;
<T_> ::= "*" <F> #4 <T_>
       | "/" <F> #5 <T_>
       | î;
<F>  ::= "(" <E> ")"
      | num #1 ;
```

Com as alterações realizadas, a legibilidade das definições aumenta, como se vê na produção abaixo:

```
<F>  ::= "(" <E> ")"
      | num #processar_num ;
```

Tal alteração, apesar da melhorar a legibilidade, não traz muitas mudanças. Porém se aliada a um recurso que facilite a recuperação de *tokens* já processados, torna-se muito

expressiva. Com base nisso, a definição da gramática foi estendida para aceitar definições de variáveis para seus símbolos.

Dessa forma para a declaração de um simbolo, terminal ou não, na especificação da gramática é possível atrelar a ela um identificador, e recupera-lo durante o processamento da ação semântica. Como exemplo:

```
<VAR_DEC> ::= a=<TIPO> b=id ";" #var_dec ;
<TIPO>    ::= "int" | "float" | "char" ;
```

Nesse ponto, o GALS ao gerar o analisador semântico irá criar uma função com o nome “var_dec” com as referências aos *tokens*, recuperados após o processamento das respectivas regras, identificados pelas variáveis descritas na gramática. Dessa forma alia-se um bom mecanismo de recuperação de *tokens*, e processamento de ações, com um contexto melhorado.

4.1.3 GRAMÁTICA DE ATRIBUTOS

Uma extensão a descrição da gramatica foi criada para permitir a elaboração de gramáticas de atributos.

Dessa forma é possível adicionar aos símbolos da gramatica, atributos para expressar seu contexto. Com tal extensão é possível, como já mencionado, adicionar atributos aos símbolos da gramática, recupera-los e compara-los.

Tal gramática é expressa da seguinte maneira, ao final das derivações de cada produção é inserido um simbolo que marca o inicio da descrição da gramática de atributos, no caso “->”. A partir desse momento pode-se inserir um conjunto de ações, separadas por um delimitador, a virgula (“,”). Essas ações são divididas em dois grupos, são eles:

- Ações de atribuição, ou seja, atribui um campo e valor a um determinado simbolo da gramática;

  Como exemplo, na sentença *int i;* pode-se atribuir ao *tokem i* um atributo “tipo”, com o valor “int”.

- Ações de comparação, compara dois campos da gramática de atributos.

  Como exemplo, na sentença *int* $a = x * y$*;* pode-se comparar o atributo “tipo” do *tokem a*, com o atributo “tipo”, resultante da derivação da expressão “x * y”.

Seu uso é simples e expressivo, como exemplo, uma pequena gramática de expressões, que faz verificação de tipo:

```
<P> ::= <E> | î;
<E>  ::= a=<T> b=<E_> -> a.type = b.type, install a.type;
<E_> ::= "+" c=<T> d=<E_> -> c.type = d.type, install c.type
         | "-" <T> <E_>
         | î -> install "type" "î";
<T>  ::= #p2 e=<F> <T_> #p3-> install e.type;
<T_> ::= "*" <F>  <T_>
         | "/" <F>  <T_>
         | î -> install "type" "î";
<F> ::= "(" <E> ")"
        | #p1 num   -> install "type" "int"
        | id -> install "type" "char" ;
```

Cabe aqui algumas considerações. Para se utilizar a gramática de atributos, tem-se que, para cada simbolo que se deseja usar, identifica-lo por uma variável, no caso do exemplo, a produção,

```
<E>  ::= <T> <E_>;
```

A qual deseja-se verificar se o tipo de “T” é o mesmo de “E_”. Assim identifica-se “T” com a variável “a” e “E_” com “b”. Em seguida faz-se a comparação. E Dessa forma,

```
<E>  ::= a=<T> b=<E_> -> a.type = b.type;
```

Em seguida cria-se o campo “type” em “E” e atribuí-se a ele o valor do campo “type” em “T”, de forma a decorar a arvore de derivação, tem-se então,

```
<E>  ::= a=<T> b=<E_> -> a.type = b.type, install a.type;
```

AÇÕES DE COMPARAÇÃO

As ações de comparações são realizadas sempre entre símbolos que compõem a produção em questão, como sugere sua árvore de derivação, em que o simbolo da produção é o nó raiz, a comparação é feita sempre entre dois de seus filhos.

É representada na gramática como a variável que identifica o simbolo da produção, seguida por um ponto (".") e o nome do campo a ser comparado, um sinal de igualdade ("=") e novamente uma variável seguida de ponto e o nome do atributo.

Dessa forma, o GALS irá gerar um analisador semântico que realizará tal comparação, e caso essa comparação seja falsa, um sinal de erro é gerado.

Algumas considerações devem ser levadas em conta, como algumas produções derivam a palavra vazia (ϵ), nesses casos pode ser que a comparação tenha que ser descartada. Para contornar tal problema, fica estabelecido que a comparação que contiver o simbolo vazio da definição do GALS ("î") é descartada.

No exemplo abaixo, anteriormente citado, a produção "T_" acaba por instalar um campo com o valor "i"

```
<T_> ::= "*" <F>  <T_>
         | "/" <F>  <T_>
         | î -> install "type" "î";
```

Dessa forma qualquer comparação que seja feita com o campo "type" do símbolo "T_", em que este derive a palavra vazia (ϵ) será descartada, ou seja não irá gerar qualquer tipo de erro.

Como exemplo, no trecho da gramática

```
<E>  ::= a=<T> b=<E_> -> a.type = b.type, install b.type;
```

para processar o símbolo "E" compara-se o campo *type* de "T" com o de "E_". Caso tais valores divirjam, um sinal de erro é gerado.

AÇÕES DE ATRIBUIÇÃO

As ações de atribuição são um tanto mais complexas, pois pode-se atribuir um valor de diversas fontes diferentes. A ação de atribuição inicia-se com "install", seguido de uma de três possibilidades de pares atributos, valores.

Como possibilidades de instalar um atributo ao simbolo da gramática tem-se,

1. Como nas ações de comparação, após ter identificado um simbolo, pode-se referencia-lo seguido de um ponto (".") e do nome do atributo, que deseja-se obter o valor. Tem-se assim o exemplo:

```
<E>  ::= a=<T> b=<E_> -> install a.type;
```

Onde, ao final do processamento, em “E” o atributo “type”, o qual possui o valor do atributo de mesmo nome de “T”.

2. Pode-se também instalar um atributo e valor diretamente pela gramática. Dessa forma, informa-se primeiro o nome do atributo, e em seguida seu valor, ambos entre aspas duplas (“ ”), como no exemplo:

```
<F> ::= num -> install "type" "int";
```

Desse processamento tem-se em “F”o atributo “type” com o valor “int”.

3. Como recurso final, ainda pode-se obter o valor de um atributo contido em uma estrutura de dados, como por exemplo, uma tabela de símbolos. Para isso, usa-se a palavra “lookup” e entre parenteses (“( )”) dois atributos, separados por virgula (“,”), onde o primeiro representa um identificador, declarado anteriormente na gramática, e o segundo o nome do atributo que se deseja recuperar. Tal qual o exemplo:

```
<F> ::= a=id -> install lookup( a, type );
```

Assim, será instalado em “F” o valor obtido pela rotina “lookup”, a qual irá procurar o valor do atributo “type” do lexema de “id”.

4.1.4 CÓDIGO FONTE GERADO

No estado atual da ferramenta, a mesma gera o código para o compilador em diversas linguagens de programação, “C++”, “Pascal” e “Java”. Porém, ao se fazer uso das extensões criadas, por hora, somente código para a linguagem “Java” é criado. Tal restrição dá-se pelo fato de que primeiro, é a linguagem que vem a ser mais utilizada pelos usuários da ferramenta; segundo, para tornar a ferramenta capaz de gerar código para outras linguagens, algumas alterações devem ser feitas. Tais alterações são possíveis, porém, por hora, fogem ao escopo deste trabalho.

Manter a compatibilidade da ferramenta, tanto entre seu uso, como no compilador gerado, mostrou-se um requisito. Por ser sua maior contribuição, seu parser, como requisito ao código gerado tem-se este, gerar um analisador semântico, compatível com o analisador sintático (parser), atualmente gerado pelo GALS.

Como a comunicação entre a fase de análise sintática e a tomada de ações semânticas, dá-se através de uma rotina específica, em que é passado ao analisador semântico o código da ação, e o último *token* processado pelo analisador léxico, tem-se então que continuar a utilizar de mesmo recurso, ou seja introduzir símbolos de ações semânticas, identificados corretamente por seu código numérico, nos estados do parser. Para dessa forma processar as ações semânticas descritas na gramática.

Dessa forma, não se compromete a estrutura interna do parser, mantem-se sua forma. E o controle das ações fica, contido no analisador semântico, sendo ele responsável por:

- Gerenciar a criação e transições de estados;
- Criar, obter ou atualizar os valores dos atributos descritos pela gramática de atributos;
- Gerenciar a tabela de atributo dos *tokens*.

A partir da especificação da gramática de atributos, e com as extensões desenvolvidas, o GALS é capaz de gerar um analisador semântico que contempla as responsabilidades acima citadas, de forma automática.

O código gerado automaticamente, para o analisador semântico, divide-se em três partes. De forma análoga ao GALS, uma classe é criada, conforme a configuração feita pelo utilizador da ferramenta, (o nome padrão da classe é *Semantico*).

A primeira parte, consiste então na declaração das estruturas de dados a serem utilizadas pelo analisador.

A segunda, trata quase que na totalidade as responsabilidades comentadas anteriormente, como o gerenciamento de estados. Toda a lógica dessa parte é realizada através da rotina "executeAction", pois, conforme comentado, tal rotina é o elo entre a análise sintática e semântica.

Em um nível mais detalhado, a rotina "executeAction" é implementada fazendo-se uso de uma estrutura de escolha (*switch-case*), para determinar o tipo de ação a ser tomada, está vindo do parser. Em cada opção de escolha uma série de rotinas são acionadas de forma a implementar as ações descritas na gramática.

Também são geradas duas rotinas de auxilio a esta parte, são elas "lookup", diretamente relacionada a gramática de atributos, e "put_token" relacionada a inserção dos *tokens*, seus atributos e valores, na tabela de símbolos, está rotina é de uso específico do usuário.

A terceira parte é responsável por gerar as rotinas de tratamento de ações especificadas pelo usuário, na descrição da gramática. Estas têm o mesmo nome especificado na gramática. Tais funções são geradas apenas com as referências aos símbolos da produção em que são inseridas, sendo elas implementadas pelo usuário.

Para dar suporte ao gerenciamento de estados uma classe é criada para representar um estado, e seus atributos. Estas classes são instanciadas pelo analisador sintático, e suas rotinas são próprias para executar as tarefas de gerenciamento de estados, de seus atributos. É através delas que as ações da gramática de atributos são implementadas. Qualquer mudança necessária para estender, ou modificar o comportamento das ações semânticas relacionadas a gramática de atributos pode ser feita através dessa classe.

BIBLIOTECA DE APOIO

Para um auxilio ainda maior ao usuário, foi criada uma biblioteca de apoio. Tal biblioteca trata especificamente do gerenciamento da tabela de símbolos, dando suporte a inserção e pesquisa de *tokens* e valores, e criação de escopo.

Essa biblioteca assemelha-se a implementação da tabelas de símbolos, sugerida por (AHO et al., 2008). Da mesma forma há o conceito de escopo global e local, e também é feito o encadeamento das estruturas, de forma a gerar uma árvore de tabelas de símbolos, onde os nós crescem no sentido do mais local ao mais global, conforme a figura 4. Dessa forma a pesquisa é feita no escopo local, e se o simbolo não for encontrado, a pesquisa passa a ser feita no nível a cima, mais global, e assim por diante, até que ou o simbolo seja encontrado, ou não haja mais tabelas.

4.1.5 TESTES REALIZADOS

O GALS Estendido tem a mesma aparência do GALS, seu funcionamento é o mesmo, portanto o desenvolvimento de um compilador, feito através do GALS continua seguindo as mesmas linhas.

Conforme descrito em (GESSER, 2003) existem três possibilidades a serem exploradas, a especificação de, somente, um analisador léxico, ou somente um analisador sintático, ou a especificação de ambos.

Para uma descrição respectivas as duas primeiras possibilidades ver (GESSER, 2003, p. 40).

O GALS Estendido difere apenas na especificação do analisador sintático, em relação

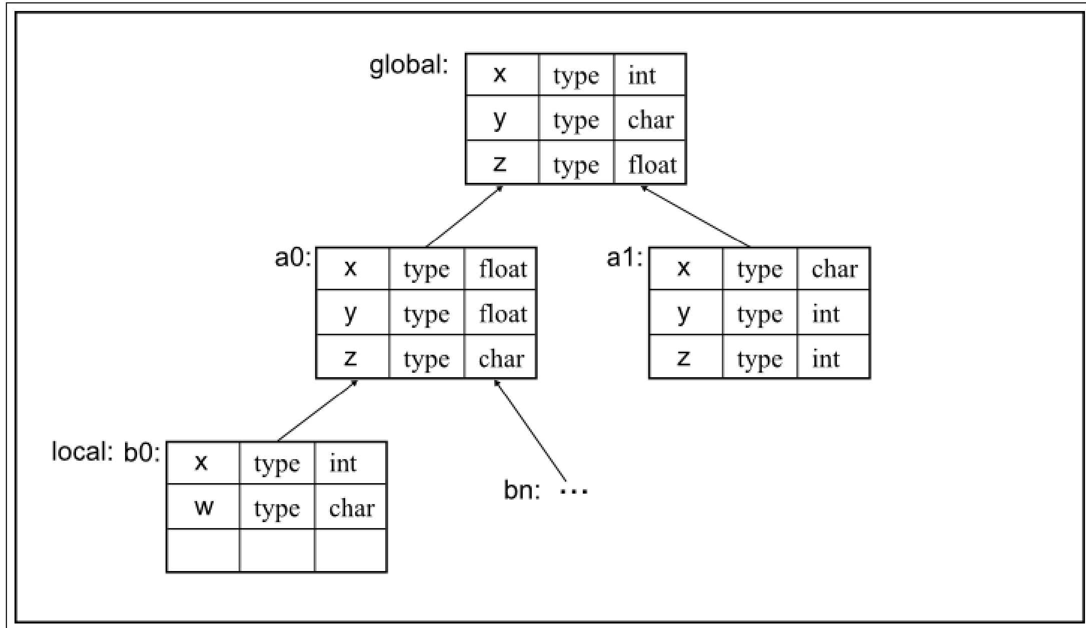

Figura 4: *Tabela de Símbolos Encadeadas*

ao GALS. Como descrito anteriormente, uma série de mudanças foram realizadas, para facilitar a fase de análise semântica.

Como exemplo a esta nova forma de declarar os aspectos sintáticos, toma-se a seguinte linguagem, descrita em BNF:

```
<P>          ::= <V> <A> <EX>;


<V>          ::= <TIPO> id ";" <V>
             |    ϵ
             ;


<A>          ::= id "=" <E> ";" <A>->
             |    ϵ
             ;


<EX>         ::= <E> ";" <EX>
             |    ϵ
             ;


<E>          ::= <T> <E_> ";" ;
```

```
<E_>          ::= "+" <T> <E_>
              |   "-" <T> <E_>
              |    ϵ
              ;

<T>           ::= "(" <E> ")"
              |   num
              |   id
              ;

<TIPO>        ::= "int"
              |   "uint"
              ;
```

Esta linguagem permite a declaração de variáveis, e realizar duas operações matemáticas, somar e subtrair, podendo fazer uso de parenteses, sobre operadores, ou sobre estas variáveis. Como exemplo, toma-se a sentença:

```
int  i;
uint j;

i = 1;
j = 10;

i + 1;
j + 100;
```

Tem-se a declaração de duas variáveis, “i” e “j”, possuindo, respectivamente, os tipos “int” e “uint”. Logo em seguida é realizado um conjunto de operações matemáticas sobre elas.

Na ferramenta desenvolvida, GALS Estendido, temos essa linguagem descrita, já na linguagem da ferramenta, em seus devidos campos, por:

Definições Regulares:

```
L : [a-zA-Z]
D : [0-9]
```

Tokens:

```
"+"
"-"
"("
")"
"="
"int"
"uint"

num : {D}+
id : {L}+
 : [" "\t\n\r]*
```

Não Terminais:

```
<P>
<V>
<A>
<E>
<E_>
<T>
<T_>
<F>
<TIPO>
```

Gramática:

```
<P>         ::= <V> <A> <EX>;

<V>         ::= tipo=<TIPO> id #install_type ";" <V>
            |   î
            ;

<A>         ::= x=id "=" y=<E> ";" <A>-> x.type = y.type
            |   î
            ;
```

```
<EX>         ::= <E> ";" <EX>
             |   î
             ;

<E>          ::= a=<T> b=<E_> ";" -> a.type = b.type, install b.type;

<E_>         ::= "+" c=<T> d=<E_> -> c.type = d.type, install c.type
             |   "-" e=<T> f=<E_> -> e.type = f.type, install f.type
             |   î -> install "type" "î"
             ;

<T>          ::= "(" <E> ")"
             |   num   -> install "type" "int"
             |   z=id -> install lookup(z, type)
             ;

<TIPO>       ::= "int"
             |   "uint"
             ;
```

Tal exemplo visa principalmente a checagem de tipo. Ao se declarar uma variável, e depois utiliza-la em uma atribuição, ou em uma expressão, seu tipo será chegado.

Como exemplo, a sentença:

```
int  i;
uint j;

i = j;
```

Irá gerar um erro semântico, pois os tipo de “i” e “j” são diferentes.

4.2 CONCLUSÕES

4.2.1 CONSIDERAÇÕES FINAIS

Dentre todas as dificuldades no processo de desenvolvimento das extensões para o GALS, o resultado atingido foi muito satisfatório. Apesar do método de Gramática de Atributos ser um dos primeiros e mais simples desenvolvidos, ele atende bem a toda uma classe de problemas, por conseguir aumentar a capacidade de reconhecimento de uma gramática livre de contexto. Entretanto a descrição da gramática, já na implementação da ferramenta, é realizada juntamente com a descrição semântica da linguagem. O ideal seria um novo componente gráfico, para que a definição da gramática ficasse separada da descrição semântica. Sugere-se então tal separação para trabalhos futuros.

4.2.2 TRABALHOS FUTUROS

Como sugestão a trabalhos futuros,

- Reformulação da interface com o usuário;
- Melhor suporte à recuperação de erros;
- Gerar imagens das construções da linguagem.

A reformulação da interface com o usuário justifica-se, como citado nas considerações finais, por ajudar o desenvolvimento da linguagem, separando as definições sintáticas das semânticas. Também pode se “aumentar” as áreas de texto, para facilitar a digitação.

Um suporte à recuperação de erros é de grande utilidade, tanto durando a construção de uma linguagem, quanto a compilação da mesma. Um mecanismo robusto de recuperação de erros é muito bem vindo, inclusive se o mesmo for capaz de apontar erros semânticos de forma clara.

Quanto a geração de imagens das construções da linguagem, é um fator acessório, porém também de grande ajuda para interpretação e análise das produções da gramática.

REFERÊNCIAS

AHO, Alfred V. et al. *Compiladores: principios, técnicas e ferramentas.* Tradução da 2. ed. [S.l.]: Pearson Addison-Wesley, 2008.

GESSER, Carlos E. *GALS - Gerador de Analisadores Léxicos e Sintáticos.* [S.l.]: UFSC, 2003.

JOHNSON, Stephen C. *Yacc: Yet Another Compiler-Compiler*. 1974. [Acesso em: 11-Julho-2009]. Disponível em: <http://dinosaur.compilertools.net/yacc/index.html>.

KNUTH, Donald E. Semantics of context-free languages. *Theory of Computing Systems*, v. 2, n. 2, p. 127–145, June 1968. Disponível em: <http://dx.doi.org/10.1007/BF01692511>.

LANG, Leonardo Trentini. *Análise de métodos para especificação semântica de linguagens de programação.* [S.l.]: UFSC, 2003.

LOUDEN, Kenneth C. *Compiladores: principios e práticas.* Tradução da 1. ed. [S.l.]: Pioneira Thomson Learning, 2004.

MENEZES., Paulo. *Linguagens formais e autômatos.* 4. ed.. ed. [S.l.]: Sagra Luzzatto, 2000.

PARR, Terence. *About The ANTLR Parser Generator*. 2007. [Acesso em: 11-Julho-2009]. Disponível em: <http://antlr.org/about.html>.

PARR., Terence. *The Definitive ANTLR Reference.* 1. ed.. ed. [S.l.]: Pragmatic Bookshelf., 2007.

SCHMIDT, Eric; LESK, Mike E. *Lex - A Lexical Analyzer Generator*. [Acesso em: 11-Julho-2009]. Disponível em: <http://dinosaur.compilertools.net/lex/index.html>.

SIPSER., Michael. *Introdução à teoria da computação.* Tradução da 2. ed. [S.l.]: Thomson Learning, 2007.