UNIVERSIDADE DE CAXIAS DO SUL

FERNANDO PAGNO DE LIMA

IMPLEMENTAÇÃO DE GERÊNCIA DE MEMÓRIA E TIPOS ESTRUTURADOS DA LINGUAGEM C NO WEBALGO

CAXIAS DO SUL - RS

2017

FERNANDO PAGNO DE LIMA

IMPLEMENTAÇÃO DE GERÊNCIA DE MEMÓRIA E TIPOS ESTRUTURADOS DA LINGUAGEM C NO WEBALGO

Trabalho de Conclusão de Curso para obtenção do título de Bacharel em Ciência da Computação pela Universidade de Caxias do Sul, orientado pelo Prof. Ricardo Vargas Dorneles.

Caxias do Sul - RS

2017

RESUMO

O uso de ferramentas automatizadas e interativas na educação está em franca expansão e, naturalmente, a área da computação acompanha essa evolução. Neste contexto da computação destacam-se os interpretadores/simuladores de linguagens de programação, muitos dos quais estão disponíveis online de forma gratuita para qualquer um que possa se interessar. A Universidade de Caxias do Sul (UCS) acompanha essa tendência com o desenvolvimento e manutenção do WebAlgo, previamente batizado de AlgoWeb. A sua versão atual combina reconhecimento e interpretação de código escrito em Português Estruturado com uma ferramenta educativa de construção e resolução de exercícios de programação. Suas funcionalidades já possibilitam o reconhecimento e interpretação de algoritmos em um pequeno subconjunto da linguagem C. Como essa linguagem é de suma importância para qualquer formação na área da computação e sendo ela a base dos sistemas operacionais mais utilizados é desejável que o WebAlgo passe a reconhecer e interpretar C para facilitar o aprendizado de novatos em programação utilizando suas ferramentas de apoio e acompanhamento. Uma questão central sobre a linguagem C e que está diretamente ligada à justificativa para este trabalho é que dificilmente será possível encontrar um programa escrito em C que não faça uso de alocação dinâmica de memória, e muito menos algum que não use ponteiros. Não se pode dizer que alguém que não seja capaz de escrever código em C que faça uso dessas duas funcionalidades tenha de fato aprendido a linguagem e esteja apto a usá-la para quaisquer fins práticos, acadêmicos ou não. Exatamente por essa razão que, após as implementações realizadas neste projeto, o WebAlgo passa a reconhecer estruturas semânticas complexas, incluindo tipos estruturados e simular o gerenciamento de memória com foco no aprendizado do usuário.

Palavras-chave: WebAlgo. Linguagem C. C. Interpretadores. Simuladores. Linguagem de Programação. Gerenciamento de Memória. Tipos Estruturados.

ABSTRACT

The use of automated and interactive tools in education is expanding considerably, and naturally, the field of computer science is keeping track with this evolution. In this context the programming languages interpreters/simulators stand out, many of which are free for use in the internet for any interested party. The University of Caxias do Sul (UCS) has been following this trend with the development and upkeep of WebAlgo, previously named AlgoWeb. Its current version combines recognition and interpretation of code written in Structured Portuguese with an educational tool for building and resolving programming exercises. Its features already enable the recognition and interpretation of a small subset of the C language. As this language is of major importance to any college level education in computer science and seeing as it is the basis of the most used operational systems in the world it is desirable that WebAlgo will come to acknowledge and interpret C as to facilitate beginner´s learning process using its monitoring and support tools. An essential issue about the C language, which is directly linked to the justification of this paper is that it will hardly be possible to find a program written in C that does not use dynamic memory allocation, much less pointers. It is not possible to say that someone who is not capable of writing code in C that uses these two features actually learned this language and will be able to use it for any practical means, academic or not. That is exactly why, after the implementations made in this project, that WebAlgo is now able to work with complex semanthic structures, including structured types and to simulate memory management focusing on the user learning experience.

Keywords: WebAlgo. C Language. C. Interpreters. Simulators. Programming Language. Memory Management. Structured Types.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ALGORITMOS

LISTA DE SIGLAS

Sigla	Significado
API	*Application Programming Interface*
ATA	*Advanced Technology Attachment*
CPU	*Central Processing Unit*
C3E	Código de Três Endereços
DLL	*Dynamic-link Library*
GB	*Gigabyte*
HD	*Hard Drive*
HTML	*Hypertext Markup Language*
IDE	*Integrated Development Environment*
JAR	*Java Archive*
JDK	*Java Development Kit*
JRE	*Java Runtime Environment*
JVM	*Java Virtual Machine*
KB	*Kilobyte*
MB	*Megabyte*
MVC	*Model View Controller*
RAM	*Random Access Memory*
SATA	*Serial Advanced Technology Attachment*
SO	Sistema Operacional
UCS	Universidade de Caxias do Sul
URL	*Uniform Resource Locator*
VM	*Virtual Machine*

SUMÁRIO

1 INTRODUÇÃO

Compilador é um programa que converte outros programas de uma linguagem de programação fonte, que via de regra é uma linguagem de alto nível na qual o programa foi escrito, para uma linguagem alvo, que geralmente é de mais baixo nível e permite ao sistema operacional executar o programa a partir de um arquivo executável (AHO, ULLMAN, & SETHI, 2008). É dessa forma que a linguagem C é utilizada.

A existência dos compiladores permite que o poder e a conveniência das linguagens de alto nível, incluindo, entre outras coisas, o suporte a orientação a objetos do Java, possa ser utilizada para construir softwares com mais qualidade e eficiência. A única outra forma de executar um código-fonte escrito em C, por exemplo, visto que um arquivo com a extensão ".c" não pode ser executado diretamente, seria executá-lo sobre um interpretador de C.

Colocado de forma simples, um interpretador é um teste de mesa automatizado, interpretando os comandos do código-fonte de forma que o funcionamento esteja de acordo com a semântica desenvolvida. Em outras palavras, funciona analogamente à forma como um programador descobriria, sem o uso de um computador, como um determinado algoritmo funcionaria em tempo de execução tendo em mãos apenas o seu código-fonte (Mak, 1996).

O Portal de Algoritmos WebAlgo da UCS, originalmente batizado de AlgoWeb foi desenvolvido combinando características de compiladores e interpretadores. O próprio Java, que é a linguagem na qual o WebAlgo foi implementado, é considerado uma linguagem híbrida pelo fato de combinar compilação e interpretação. O compilador *javac* converte o código-fonte Java para uma representação intermediária chamada *bytecode*, localizada dentro dos arquivos *class*, que é interpretada pela JVM (WEBER, 2016, pp. 30-31).

De fato, muitas das estruturas utilizadas para representações internas e intermediárias são comuns aos dois como evidenciado em (Mak, 1996). O WebAlgo contém um editor orientado à sintaxe de Português Estruturado, que recentemente foi expandido para reconhecer um micro-conjunto da linguagem C e também possui suporte à depuração com processamento passo-a-passo, pontos de parada e monitoramento de variáveis. Também possui um banco de problemas algorítmicos e realiza verificação automática de correções dos algoritmos submetidos para cada

problema através do uso de casos de testes desenvolvidos com entrada e saída de dados pré-definidas (DORNELES, JUNIOR, & ADAMI, 2011).

Sendo C uma linguagem que trabalha frequentemente com alocação dinâmica de memória aparece a necessidade da pesquisa relacionada ao gerenciamento de memória, definido como a gerência da hierarquia de memória de um computador. Hierarquia de memória é uma abstração que envolve quase a totalidade dos espaços de memória disponíveis no computador, incluindo cache, memória principal e espaço em disco, e existe para fornecer um modelo de utilização útil para o uso dos processos. Gerenciamento de memória é uma das atividades fundamentais de qualquer SO, realizada através do seu gerenciador de memória. Seu trabalho é observar e saber quais partes da memória estão ocupadas ou livres, alocar memória para processos quando solicitado e liberá-la quando os processos finalizam sua execução (TANENBAUM, 2007).

1.1 PROBLEMA DE PESQUISA

Atualmente o departamento de informática da área de ciências exatas da Universidade de Caxias do Sul já conta com um simulador que combina um compilador/interpretador que reconhece e interpreta Português Estruturado com uma ferramenta educativa de construção e resolução de exercícios de programação. Recentemente ele recebeu o nome de WebAlgo, apesar de sua última versão ter sido convertida para desktop. Apesar do fato de ser de grande valia para os alunos de computação não se trata de uma idéia inovadora, pois já existem ferramentas muito semelhantes em diferentes contextos, como por exemplo o do site UVA Online Judge (https://uva.onlinejudge.org/).

As versões mais recentes do WebAlgo possuem a nova funcionalidade que permite desenvolver e interpretar algoritmos na linguagem C. É de conhecimento comum a toda a comunidade acadêmica de computação que o aprendizado dessa linguagem é de suma importância para qualquer formação na área, sendo ela a base de todos os grandes sistemas operacionais e a precursora das linguagens mais utilizadas no mundo.

Dificilmente será possível encontrar um programa escrito em C que não faça uso de alocação dinâmica de memória, e muito menos algum que não use ponteiros. Não se pode dizer que alguém que não seja capaz de escrever um programa em C

que faça uso dessas duas funcionalidades tenha de fato aprendido a linguagem e esteja apto a usá-la para quaisquer fins práticos, acadêmicos ou não.

Exatamente por essa razão que, após as implementações previstas para este projeto, o WebAlgo deverá reconhecer estruturas semânticas complexas tais como:

*& ((*inicio)->prox)*,

o endereço do membro “prox” da *struct* apontada por “*inicio”, ou então:

cliente[i].campo1[j].campo->sei_la,

o valor do campo “sei_la”, localizado na *struct* apontada pelo ponteiro “campo”. O operador “->” é usado para acessar diretamente um campo de uma *struct* a partir de um ponteiro para a mesma. O ponteiro “campo”, por sua vez, aponta para uma *struct* contida dentro de outra *struct* contida na posição “j” de um vetor chamado “campo1”, que também está dentro de outra *struct* que por sua vez também está contida dentro de um vetor, especificamente na posição “i”.

Portanto é desejável incrementar essa solução para que os estudantes possam melhorar o seu aproveitamento das disciplinas de programação oferecidas pelo curso. Para tanto é necessário que ele passe a reconhecer uma parte significativa da linguagem e que possa trabalhar com estruturas complexas que envolveriam alocação dinâmica de memória, como as *structs* e passe a reconhecer e simular a funcionalidade de ponteiros. Atualmente o programa já tem todos os fluxos de controle implementados em C, além de vetores e matrizes, o que também contempla *strings*.

1.2 QUESTÃO DE PESQUISA

Qual a forma mais eficaz de implementar o reconhecimento de estruturas complexas como *structs* e ponteiros, além de alocação dinâmica de memória, da linguagem C ao já existente compilador/interpretador didático utilizado nas disciplinas de algoritmos e programação?

1.3 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é propor e desenvolver um projeto para o desenvolvimento de um simulador da linguagem C dentro do WebAlgo que seja tão funcional quanto o já existente em Português Estruturado permitindo trabalhar com os tipos estruturados da linguagem.

Para atingir o objetivo mencionado, foram definidos os seguintes objetivos específicos, a fim de que sejam subprodutos desse trabalho:

a) Definir estruturas de dados para suportar as estruturas complexas
b) Projetar o gerenciamento da memória para simular alocações dinâmicas do C
c) Elaborar uma documentação básica da estrutura de classes e padrões de projeto do WebAlgo

1.4 ESTRUTURA DO TRABALHO

O presente trabalho está estruturado da seguinte forma: no Capítulo 2 são apresentadas informações sobre o referencial teórico do projeto, as ferramentas que serão utilizadas e os seus objetos de estudo e pesquisa. Isso inclui um estudo de como ele está estruturado no momento. No Capítulo 3 são apresentadas as características técnicas do WebAlgo, analisadas através de um método particular de engenharia reversa que permita realizar uma análise sucinta do seu funcionamento. O Capítulo 4 descreve as estruturas de dados propostas para atender à necessidade descrita na questão de pesquisa e no objetivo do trabalho, além das funcionalidades de simulação de gerenciamento de memória.

O capítulo 5 descreve a forma como o projeto foi desenvolvido na sua etapa de implementação. As descrições das funcionalidades efetivamente implementadas, incluindo o uso que foi feito no desenvolvimento das propostas descritas no capíutulo 4, estão expostas no capítulo 6. No capítulo 7 são descritos os testes mais importantes aplicados sobre a versão do programa desenvolvida neste projeto.

E, finalmente, o capítulo 8 apresenta as conclusões do trabalho.

2 REFERENCIAL TEÓRICO

A seguir estão descrições do embasamento teórico envolvido no desenvolvimento do trabalho, partindo dos conceitos de compiladores e interpretadores de linguagens de programação. Consiste em conceitos da área de estudo delimitada somados a assuntos tangenciais que aumentam o escopo de pesquisa, como o gerenciamento de memória, além da apresentação das ferramentas utilizadas e suas características mais relevantes para o projeto.

2.1 COMPILADORES E INTERPRETADORES

Qualquer computação realizada por um computador em algum momento deve ser traduzida para a linguagem de máquina e finalmente executada pelo hardware sobre o qual o software está sendo executado. Portanto, qualquer comando previsto em um programa de computador, quando processado pelo sistema operacional em funcionamento, vai resultar em uma computação que deve ser executada pelo hardware gerenciado pelo SO. O significado disso é que algoritmos escritos em linguagens de programação de alto nível, que fornecem abstrações muito úteis para aproximar a complexidade do hardware das capacidades lógicas do programador, devem em algum momento ser traduzidos para linguagem de máquina. Sob essa ótica é possível afirmar que a computação como a conhecemos hoje não seria possível sem a existência de compiladores e interpretadores (AHO, ULLMAN, & SETHI, 2008).

Compilador é um programa que converte outros programas de uma linguagem de programação fonte, que via de regra é uma linguagem de alto nível na qual o programa foi escrito, para uma linguagem alvo, que geralmente é de mais baixo nível e permite ao sistema operacional executar o programa a partir de um arquivo executável (AHO, ULLMAN, & SETHI, 2008).

Uma definição mais completa é fornecida em (Mak, 1996):

> A compiler translates a program written in the source language into a low-level object language, which can be the assembly language or the machine language of a particular computer. The program that you write in the source language is called the source program, which you edit in one or more source files. The compiler translates each source file into an object file. If the object files contain assembly language you must next run an assembler (yet another type of program translator) to convert them into machine language. You then run a utility program called a linker to combine the object files (along with any needed runtime library routines) into the object program. Once created, an

> object program is a separate program in its own right. You can load it into the computer´s memory and then execute it.
> (Mak, 1996, p. 2)

A citação especifica funções posteriores ao processo de compilação que precisariam ser executadas para obter um código objeto, que pode ser carregado para a memória do computador e executado diretamente, por ter suas instruções expressas em código de máquina. Para os propósitos deste trabalho desconsidera-se o processo que converte para linguagens de baixo nível. Considera-se compilado o programa que foi convertido para um arquivo executável, podendo ser executado diretamente pelo SO.

Um interpretador é um teste de mesa automatizado, interpretando os comandos do código-fonte de forma que o funcionamento esteja de acordo com a semântica desenvolvida. Em outras palavras, funciona analogamente à forma como um programador descobriria, sem o uso de um computador, como um determinado algoritmo funcionaria em tempo de execução tendo em mãos apenas o seu código-fonte, de acordo com a segunda parte da comparação:

> On the other hand, an interpreter does not produce an object program. It may translate the source program into an internal intermediate code that it can execute more efficiently, or it may simply execute the source program's statements directly. The net result is that an interpreter translates a program into the actions specified by the program.
> (Mak, 1996, p. 2)

Quando se trata de decidir qual é a melhor opção entre um compilador e um interpretador, analisando suas vantagens ou desvantagens, depara-se com um dilema comum na computação: depende da aplicação.

O interpretador tem maior versatilidade e poder de depuração em tempo de execução. Programadores Web que utilizam a linguagem JavaScript, por exemplo, sabem bem disso, pois os navegadores web são todos interpretadores que trabalham sobre os documentos HTML e as linguagens de *scripting*, o que significa que possuem ferramentas que permitem ao programador depurar todo o estado do HTML e todos os *scripts* utilizados em tempo de execução. Além disso, geralmente o interpretador permite que o programa seja executado parcialmente mesmo que em outros trechos de código ele contenha erros de sintaxe, atribuições inválidas e falha na declaração de alguma variável.

Já os compiladores exigem um esforço extra até que se possa executar o programa, pois exige que ele passe pelo processo de compilação para gerar o programa alvo com base no programa fonte, e além disso não permitem depuração em um ambiente de produção, já que o que está sendo executado é apenas o código objeto ou o executável. Mas essa realidade está mudando, uma vez que os compiladores embutidos em IDEs possuem ferramentas avançadas de depuração e execução passo-a-passo, além de permitirem que um programa seja executado diretamente através da sua IDE.

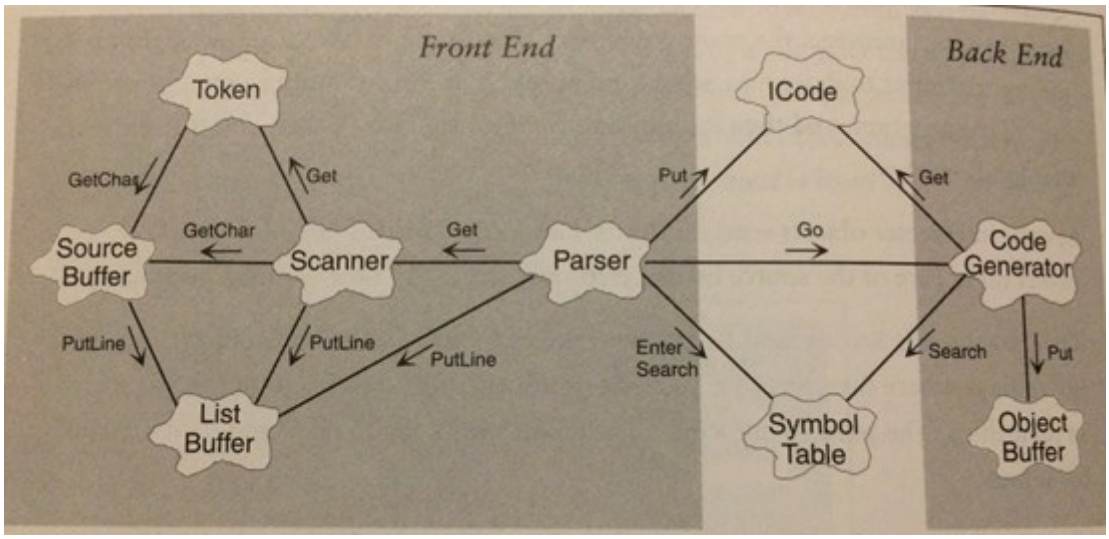
E também é importante lembrar que um programa gerado a partir de um compilador pode estar otimizado devido a capacidades embutidas no próprio processo de compilação. E mesmo desconsiderando essa funcionalidade progressivamente mais comum em compiladores modernos, um programa compilado ainda assim tem a vantagem da performance ao seu lado, pois trata-se de um programa executado de forma independente (apenas um processo no SO), enquanto no caso de interpretador trata-se de um programa executando sobre outro (pelo menos dois processos concorrentes) e, portanto, exige maior processamento.

2.1.1 Semelhanças e Diferenças Estruturais

Apesar de terem funções diferentes, estruturalmente os compiladores e os interpretadores podem ser muito similares. De fato, conforme destacado em (Mak, 1996), de acordo com os dois diagramas abaixo a única diferença está na parte que o autor nomeou como *back end*, ou os objetos desenvolvidos para realizar a parte final do processo.

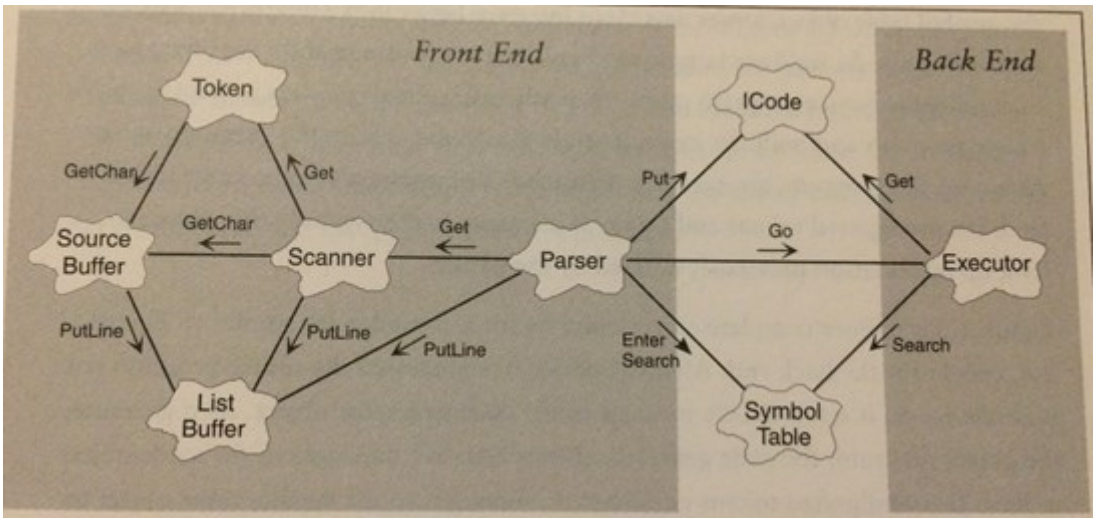
Pela análise resultante da comparação das Figuras 1 e 2 o autor evidencia por que o estudo dos dois tipos de software cabe em um único tópico e também como seria possível desenvolver um programa que atuasse ora como um compilador e em outros momentos como um interpretador, supondo que existam necessidades ou requisitos que exijam esse tipo de versatilidade.

Figura 1 – Diagrama de Objetos de um Compilador



Fonte: (Mak, 1996, p. 10)

Figura 2 – Diagrama de Objetos de um Interpretador



Fonte: (Mak, 1996, p. 9)

Basicamente, o *front end* lê o programa fonte e o *back end* executa o programa ou gera o seu equivalente em código objeto ou arquivo executável, dependendo se foi desenvolvido para a função de compilador ou interpretador.

O *back end* especificado por (Mak, 1996) equivale às fases de síntese de um compilador, conforme definido em (AHO, ULLMAN, & SETHI, 2008), supondo que o programa em questão não seja um interpretador, em cujo caso não existe a síntese do programa objeto. O *front end* equivale à fase de análise.

No diagrama de objetos da Figura 2 há uma linha conectando dois objetos, cada um representado na forma de uma nuvem, revela que existe troca de mensagens entre eles. Há uma seta indicando a direção da mensagem, do remetente para o destinatário e uma descrição sobre a mesma com o nome do método chamado que indica a ação para a qual a conexão existe.

Analisando a arquitetura apresentada conclui-se que a análise léxica bem como a sintática são realizadas pelo *front end* do programa. O objeto *Parser*, possuindo o conhecimento da sintaxe da linguagem fonte, controla o processo. Ele solicita ao objeto *Scanner* o próximo *token* a ser processado e realiza a análise sintática baseada na gramática que ele inerentemente conhece e no contexto dos *tokens* lidos anteriormente. Simultaneamente ele gera o código intermediário (*ICode*, ou *intermediate code*) e a tabela de símbolos (*Symbol Table*), que são as duas estruturas que servem como uma interface com o *back end*, permitindo a alternância entre interpretação e compilação.

O *Scanner* é o objeto que realiza o trabalho de análise léxica, ou seja, separar o código fonte nas diferentes palavras e símbolos que o compõem, cada um dos quais é denominado *token* e é representado através de uma nova instância do objeto *Token*. Cada um destes pode ser um identificador (variável), uma palavra reservada que representa ou faz parte de um comando ou função específica da linguagem, ou ainda pode ser um valor determinado para alguma variável ou parâmetro. Os caracteres são buscados sequencialmente a partir do objeto *Source Buffer*, que por sua vez os busca no arquivo-fonte. Cabe ao objeto *Token* determinar qual tipo de *token* a nova instância representa e repassá-lo para o *Scanner* e, indiretamente, para o *Parser*.

Todos os identificadores encontrados serão inseridos pelo *Parser* dentro da *Symbol Table* assim permitindo que ele faça o controle de declaração e de tipos de variáveis. Quaisquer erros léxicos ou sintáticos encontrados são inseridos no objeto *List Buffer* para que sejam tratados posteriormente. Ele então combina os identificadores, valores e comandos encontrados para determinar o código intermediário equivalente ao código fonte e insere o mesmo em *ICode*.

O *back end* realiza o trabalho de interpretação ou de compilação a partir da tabela de símbolos e do código intermediário. O interpretador apenas executa o código intermediário enquanto o compilador usa-o para gerar o código objeto ou executável (Mak, 1996). Todas as classes instanciadas nos diagramas estão codificadas na

íntegra na fonte citada, desenvolvidas na linguagem C++ para trabalhar com código-fonte desenvolvido em Pascal.

2.1.2 Tradução Dirigida por Sintaxe

Tradução dirigida por sintaxe é uma técnica de compilação orientada pela gramática. O objetivo é traduzir instruções de uma linguagem de programação representativa para uma ou mais representações intermediárias. Algumas destas podem ser apenas parciais no sentido de que não refletem a íntegra da semântica descrita no fonte, mas devem ser necessariamente funcionais no sentido de agregar eficiência ou qualidade ao código objeto ou interpretação (AHO, ULLMAN, & SETHI, 2008).

Todas as ferramentas descritas a seguir fazem parte da fase de análise, ou *front end*. Todo o processo é organizado em torno da sintaxe da linguagem, que é especificada através da notação chamada gramática livre de contexto.

A estrutura de um comando condicional *if*, usado em muitas linguagens é a seguinte:

if (**exp**) **com** else **com**

No lugar de *exp* deve haver uma expressão lógica cujo valor avaliado em tempo de execução define se o programa entra no trecho de código após o *if* ou após o *else*, de acordo com a estrutura padrão de um comando condicional. E no lugar de *com* deve haver um comando, ou então uma lista deles, que será executado ou não de acordo com a avaliação da expressão condicional.

Ao definir uma gramática para especificar essa sintaxe utiliza-se a estrutura abaixo, denominada produção. Essa produção indica que o condicional *if* é um comando dentro da linguagem e determina qual a sua estrutura.

com -> if (**exp**) **com** else **com**

Os elementos léxicos da produção que não estão marcados em negrito são chamados de terminais. Estes são os símbolos elementares da linguagem.

O que está escrito em negrito são as variáveis não-terminais, cada uma das quais deve gerar uma ou mais produções dentro da gramática especificando a sua estrutura sintática. Na prática isso significa que, durante o reconhecimento do código fonte, elas serão substituídas repetidamente pelo corpo de uma de suas produções até que a derivação resulte nos exatos lexemas existentes no programa fonte. É dessa forma que a sintaxe do programa é verificada.

O não-terminal *exp* deverá ter produções que determinem os seus possíveis formatos gramaticais e *com* deverá ter mais produções especificando todos os outros comandos da linguagem, podendo inclusive derivar listas de comandos.

Toda gramática deve ter um símbolo inicial, que é um não-terminal, a partir do qual se inicia a derivação. As cadeias de terminais que podem ser derivadas a partir dele formam a linguagem descrita pela gramática.

A tradução dirigida por sintaxe ocorre através da execução de regras ou fragmentos de código durante o processo de derivação das produções. Cada produção derivada executa diferentes ações com o objetivo de gerar estruturas intermediárias representativas do código fonte.

A mesma técnica pode ser utilizada para traduzir expressões de uma notação para outra, usando esquemas de tradução e atributos relacionados aos não-terminais. A definição abaixo traduz uma expressão infixada para notação pós-fixada.

9 – 5 + 2 notação infixada (<operando> <operador> <operando>)

9 5 – 2 + notação pós-fixada (<operando> <operando> <operador>)

Tabela 1 - Definição para tradução de expressão infixada para pós-fixada

PRODUÇÃO	REGRAS SEMÂNTICAS
expr -> expr1 + term	expr.t = expr1.t || term.t || ‘+’
expr -> expr1 – term	expr.t = expr1.t || term.t || ‘-‘
expr -> term	expr.t = term.t
term -> 0	term.t = ‘0’
term -> 1	term.t = ‘1’
...	...
term -> 9	term.t = ‘9’

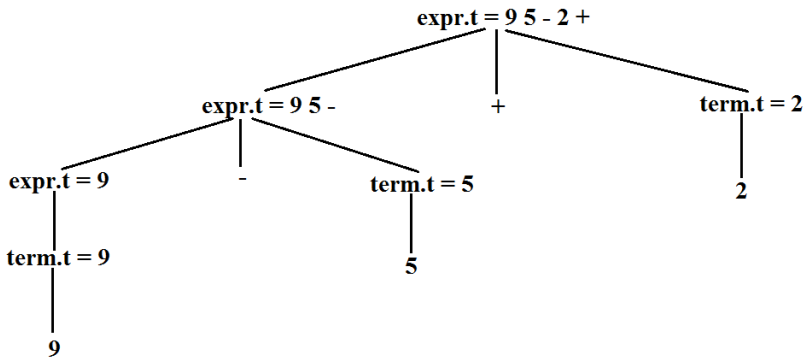
Fonte: (AHO, ULLMAN, & SETHI, 2008, p. 35)

Na Tabela acima cada não-terminal possui o atributo *t* de valor literal que representa a notação pós-fixada para a expressão gerada por esse não-terminal durante a derivação. O símbolo || na regra semântica é o operador de concatenação das cadeias de caracteres.

Nas duas primeiras produções da Tabela 1 a regra semântica gera uma expressão pós-fixa realizando a concatenação dos dois operandos antes de concatenar o operador no final, sendo que o não-terminal que representa o primeiro operando poderá ter derivado uma expressão pós-fixada.

A Figura 3 ilustra o esquema de derivação para a definição da Tabela 1 em uma árvore anotada usando como exemplo a expressão 9 – 5 + 2 especificada acima.

Figura 3 – Árvore de derivação anotada para a definição da Tabela 1



Fonte: (AHO, ULLMAN, & SETHI, 2008, p. 35)

2.1.3 Tabela de Símbolos

A tabela de símbolos é uma estrutura construída pelo compilador ou interpretador para o seu próprio uso ao longo do seu trabalho. É a estrutura de dados usada para conter informações sobre construções encontradas no programa fonte, via de regra, os identificadores. Os registros inseridos na tabela contém informações tais como o nome (ou lexema) do identificador, seu tipo, contexto em que foi declarado, e qualquer outra informação sobre eles que possa ser relevante para alguma das etapas do processo (AHO, ULLMAN, & SETHI, 2008). As construções encontradas dentro do fonte são comumente designadas, a partir da análise léxica, como *tokens*.

Outros *tokens*, que não se caracterizam como identificadores, também podem ter seus próprios registros na tabela, dependendo das funcionalidades e particularidades do programa em questão.

Como observado anteriormente, nas Figuras 1 e 2, a tabela de símbolos é um componente imprescindível para a interface entre o *front* e o *back end* (Mak, 1996). Durante a tradução do programa fonte o compilador/interpretador deve ser capaz de coletar, de forma incremental, as informações necessárias para cada *token*, bem como atualizar e consultá-las na tabela de símbolos de forma rápida e eficiente. Do contrário, o processo pode se tornar lento ou mesmo ineficaz.

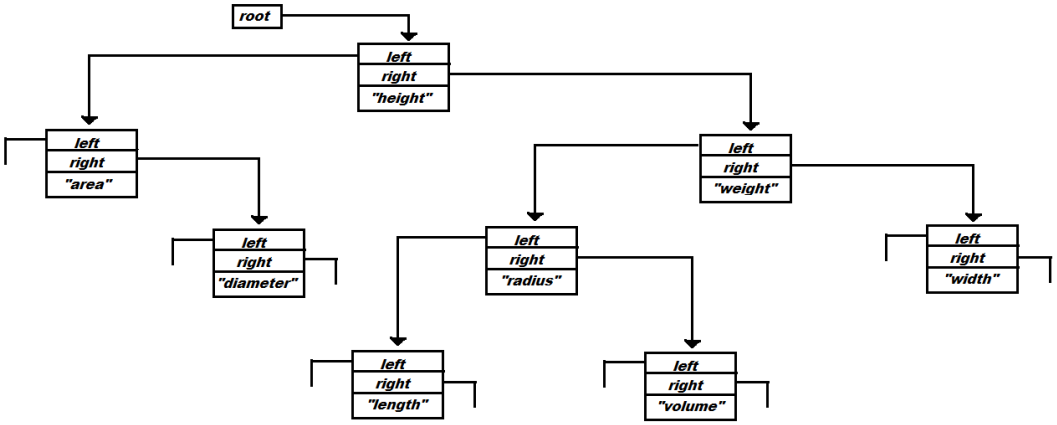
Existem diversas estruturas de dados capazes de representar uma tabela de símbolos, tais como listas encadeadas, árvores, *hash tables* e até mesmo vetores. Para escolher qual delas é a mais adequada é preciso se basear em quantas informações ela vai conter (o número de campos) e aproximadamente quantos símbolos ela poderá conter em tempo de execução (número de registros).

Considerando o fato de que as operações realizadas sobre a tabela de símbolos durante a sua construção são inserção e consulta de *tokens* e também o fato de ela estar perfeitamente ordenada melhora muito o desempenho das pesquisas então é uma boa solução organizá-la como uma árvore binária (Mak, 1996). A Figura 4 mostra uma tabela de símbolos simples organizada como uma árvore binária (*binary tree*), iniciando a partir de um nodo raiz (*root*), que em uma implementação em código seria apenas um ponteiro para o nodo com o identificador *height* abaixo.

A forma como as árvores binárias são organizadas consiste na noção de que cada nodo pai está no meio dos seus dois nodos filhos, para os quais ele tem dois ponteiros, de acordo com a ordenação aplicada sobre a árvore, que neste caso é a ordem alfabética. O *token* que está no nodo à esquerda do pai sempre está antes dele no alfabeto. E o filho da direita terá um *token* posterior ao pai.

A organização em árvores binárias torna as consultas muito rápidas e pode ser aplicada em muitos outros casos na computação. Na maioria das implementações a tabela de símbolos teria de conter mais informações do que apenas os identificadores das variáveis encontradas. Cada nodo teria que informar em qual linha ela foi declarada, em que contexto (qual função) e qual o seu tipo.

Figura 4 – Uma Tabela de Símbolos organizada em uma Árvore Binária



Fonte: (Mak, 1996)

Tipicamente uma tabela de símbolos precisa dar suporte a diversas declarações do mesmo identificador, contanto que estas estejam dentro de contextos diferentes. Esses contextos se referem a diferentes funções ou procedimentos dentro de um mesmo programa fonte, e também podem ser entendidos como escopos. Se uma variável está declarada dentro de uma função então o seu uso só é válido dentro daquela função e ela não pode ser referenciada dentro de outra função, incluindo a função ou método principal a partir do qual se inicia a execução do programa. Se, por outro lado, o identificador em questão estiver declarado dentro do escopo de duas funções então ele é válido nos dois contextos, embora as duas instâncias representem locais de memória distintos, o que significa que são duas variáveis diferentes (AHO, ULLMAN, & SETHI, 2008).

Controlar qual das duas variáveis está sendo referenciada, baseando-se no escopo analisado em um determinado momento em tempo de execução, é um trabalho que depende da tabela de símbolos.

Uma possível abordagem é criar uma tabela para cada escopo. Todos os procedimentos e funções terão a sua própria tabela de símbolos com um registro para cada identificador especificado na sua seção de declaração. Se a linguagem fonte for orientada a objetos, e supondo uma solução semelhante, então cada campo, propriedade e método terá o seu próprio registro na tabela de símbolos. E cada um dos métodos da classe tem o seu contexto.

> O termo “escopo do identificador x” na realidade se refere ao escopo de uma declaração particular de x. O termo escopo por si só refere-se a uma parte de um programa que é o escopo de uma ou mais declarações.

> Os escopos são importantes, pois o mesmo identificador pode ser declarado para diferentes finalidades em diferentes partes de um programa. Nomes comuns como i e x têm usualmente múltiplos usos. Outro exemplo, as subclasses podem redeclarar um nome de método que redefine um método de uma superclasse.
> Se os blocos puderem ser aninhados, várias declarações do mesmo identificador podem aparecer dentro de um único bloco.
> Fonte: (AHO, ULLMAN, & SETHI, 2008)

Para determinar a qual escopo uma variável x pertence, em um contexto de vários blocos aninhados, pode-se utilizar a regra do aninhamento mais interno. Consiste em analisar a partir do bloco no qual x é utilizada e buscando as declarações em contextos progressivamente mais externos até encontrar a primeira declaração de x. A regra pode ser implementada encadeando tabelas de símbolos, com a tabela de um bloco aninhado apontando para a tabela do bloco que o envolve. Dessa forma a navegação entre tabelas de símbolos torna-se relativamente fácil e eficiente.

O esquema da Figura 5 demonstra, usando subscrito, a distinção entre declarações de identificadores iguais em diferentes contextos aninhados, onde o número ao lado da letra que identifica a variável declarada não faz parte do seu identificador, mas demonstra a qual bloco pertence a sua declaração ou o seu uso.

Na primeira linha, onde o contexto mais externo começa, são declarados **x** e **y**. Na segunda linha **y** é declarado novamente em meio a outras variáveis declarados no bloco mais interno, o que significa que qualquer uso de **y** dentro do bloco contido entre as linhas 2 e 4 será o uso da variável declarada no contexto mais interno, como evidenciado no seu uso na linha 3.

Todas as ocorrências de **x** estão dentro do escopo da declaração na linha 1. A ocorrência de **w** na linha 5 não pode referenciar a variável declarada na linha 2 pois não está inserida naquele escopo. Portanto está provavelmente dentro do escopo de uma declaração de **w** que está fora do fragmento de código analisado, ou seja, ainda mais externo do que o bloco iniciado na linha 1. Ela também pode ter sido declarada de forma global, onde seu subscrito 0 indica declaração global ou externa.

Portanto, a Figura 5 deve ser analisada supondo que os números das variáveis não fazem parte do seu identificador, sendo apenas indicadores que determinam a qual escopo de declaração elas pertencem.

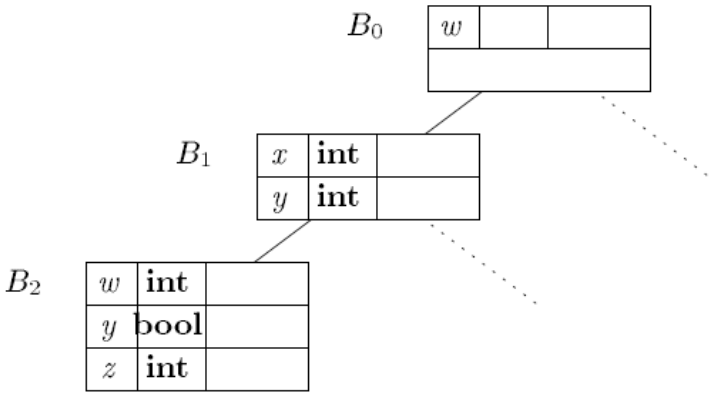
Figura 5 – Esquema de aninhamento de declarações e uso de variáveis

```
1)   {     int x1;   int y1;
2)         {     int w2;   bool y2;   int z2;
3)               ... w2 ...;     ... x1 ...;     ... y2 ...;     ... z2 ...;
4)         }
5)         ... w0 ...;     ... x1 ...;     ... y1 ...;
6)   }
```

Fonte: (AHO, ULLMAN, & SETHI, 2008, p. 56)

Dessa forma, considerando os blocos de código e as variáveis identificadas no fragmento da Figura 5 pode-se implementar a regra de aninhamento mais interno pelo encadeamento de tabelas de símbolos, como evidenciado na Figura 6.

Figura 6 – Encadeamento das tabelas de símbolos para o esquema da figura 5



Fonte: (AHO, ULLMAN, & SETHI, 2008, p. 57)

2.1.4 Código Intermediário

Conforme mencionado anteriormente, o papel do *front end* é analisar o programa fonte e gerar representações intermediárias, entre elas a tabela de símbolos, a partir das quais o *back end* gera o código objeto. Os detalhes da linguagem fonte devem ser confinados no *front end* e os detalhes da máquina alvo, para a qual será gerado o código objeto, devem estar encapsulados no *back end* (AHO, ULLMAN, & SETHI, 2008, p. 228).

O objetivo da tabela de símbolos é reunir informações sobre os identificadores que aparecem no programa, para poder utilizá-las durante e após a etapa de análise.

As representações intermediárias do programa fonte, também construídas durante a etapa de análise (pelo *front end*), possibilitam a própria separação entre front e *back end*. São usadas pelo compilador para sintetizar o programa objeto. E o código intermediário é o que o interpretador, potencialmente com o auxílio de outras estruturas, de fato interpreta para produzir os resultados esperados pelo programador do código fonte (Mak, 1996).

Uma representação intermediária é um tipo de linguagem de máquina abstrata, resultante da fase de análise semântica, que é capaz de expressar as operações para uma máquina-alvo, que é o sistema para o qual o programa está sendo compilado, sem, contudo, conter muitos detalhes específicos e determinantes para a máquina em questão. Ao mesmo tempo ela deve estar totalmente livre das particularidades da linguagem fonte sobre a qual o *front end* realizou o seu trabalho. O *back end* trabalha sobre as representações intermediárias, otimizando-as e traduzindo-as para a linguagem da máquina-alvo (Appel, 1999).

As duas representações básicas mais importantes são as árvores, tanto as de derivação como também as sintáticas, e as representações lineares, ou em código, tais como o código de três endereços, comumente abreviado como C3E. O C3E é uma representação linear do programa, sem estruturas hierárquicas, na qual todas as operações realizadas pelo fonte, por mais complexas que possam ser, estão traduzidas em operações triviais de aritmética, atribuição ou desvios condicionais e incondicionais (AHO, ULLMAN, & SETHI, 2008). Para a realização de qualquer tipo de otimização de código o C3E, ou outra representação equivalente, é fundamental.

Algoritmo 1 - Um fragmento de código a ser traduzido

```
int i; int j; float[100] a; float v; float x;

while (true) {
      do i = i + 1; while (a[i] < v);
      do j = j - 1; while (a[j] > v);
      if ( i >= j ) break;
      x = a[i]; a[i] = a[j]; a[j] = x;
}
```

Fonte: (AHO, ULLMAN, & SETHI, 2008)

Algoritmo 2 – Código intermediário simplificado para o fragmento do algoritmo 1

```
1:    i = i + 1
2:    t1 = a [ i ]
3:    if t1 < v goto 1
4:    j = j – 1
5:    t2 = a [ j ]
6:    if t2 > v goto 4
7:    ifFalse i >= j goto 9
8:    goto 14
9:    x = a [ i ]
10:   t3 = a [ j ]
11:   a [ i ] = t3
12:   a [ j ] = x
13:   goto 1
14:
```

Fonte: (AHO, ULLMAN, & SETHI, 2008)

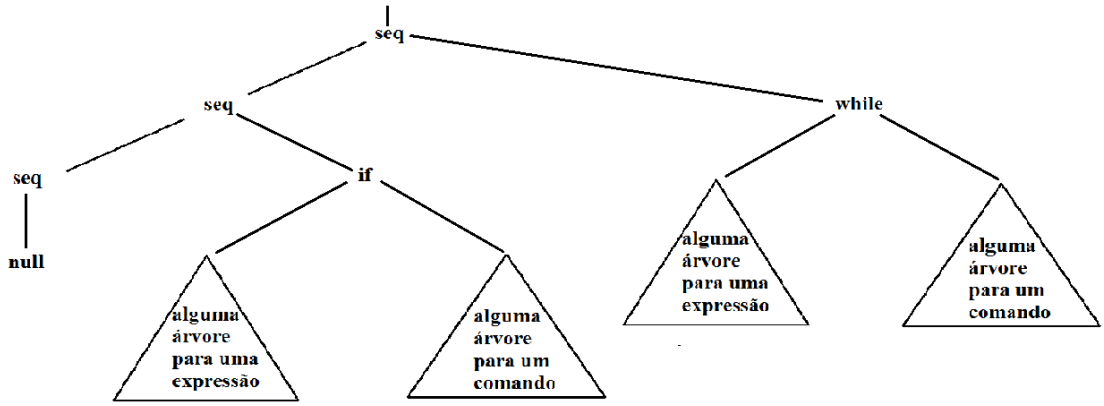
> É possível que um compilador construa uma árvore sintática ao mesmo tempo que emite trechos do código de três endereços. Contudo, é comum que os compiladores emitam o código de três endereços enquanto o analisador “se movimenta” construindo uma árvore sintática, sem realmente construir a estrutura de dados completa da árvore. Em vez disso, o compilador armazena nós e seus atributos necessários para a verificação semântica ou outras finalidades, junto com a estrutura de dados usada para análise. Fazendo isso, as partes da árvore sintática necessárias para construir o código de três endereços estão disponíveis quando necessárias, mas desaparecem quando não são mais necessárias.
> Fonte: (AHO, ULLMAN, & SETHI, 2008)

Árvores sintáticas são usadas para representar expressões formadas pela aplicação de um operador a duas subexpressões, que não são necessariamente expressões inteiras, e que podem ser gramaticalmente traduzidas apenas em um identificador ou outro *token* escrito diretamente no código (*hard coded*).

Árvores sintáticas também podem ser usadas para representação de outros tipos de construções da linguagem fonte como comandos condicionais ou de iteração. Por exemplo, um comando *while* em C é formado por dois componentes semanticamente significativos, que são: a expressão analisada para a decisão de permanecer ou não dentro do laço e o bloco de comandos executado dentro do laço. Portanto o *while* seria traduzido para uma árvore sintática como uma subárvore na qual o *while* é a raiz tendo os seus dois componentes como filhos: a expressão condicional à sua esquerda e o bloco de comandos à sua direita, cada um dos quais também podendo ser representado por uma subárvore.

A mesma lógica também pode ser aplicada ao comando condicional *if*, que terá a mesma subestrutura em uma representação em árvore. Uma sequência de comandos na árvore é representada por uma folha *null* para um comando vazio e um operador *seq* para uma sequência de comandos, como evidenciado na Figura 7.

Figura 7 – Árvore sintática para lista de comandos consistindo de if e while



Fonte: (AHO, ULLMAN, & SETHI, 2008, p. 62)

O C3E e as árvores sintáticas de comandos representando expressões podem ser gerados ao mesmo tempo dependendo de como o esquema de tradução foi estruturado. As árvores sintáticas representando listas de comandos podem ser dispensadas enquanto as regras de tradução apenas geram o C3E, entretanto o uso de árvores para avaliação de expressões é altamente recomendável.

O C3E pode conter apenas operações envolvendo um operador e dois operandos em uma atribuição, além de desvios condicionais e incondicionais. Nenhuma construção aritmética construída com vários operadores é permitida (AHO, ULLMAN, & SETHI, 2008, p. 232). Abaixo estão representados exemplos das construções aceitas pela linguagem.

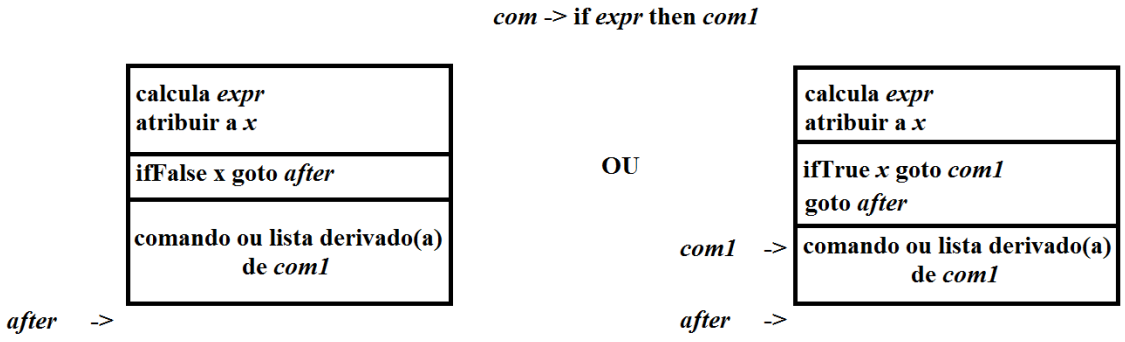
x = y op z	**x = y [z]**	**ifFalse x goto L**
	x [y] = z	**ifTrue x goto L**
		goto L

Cada linha também pode ter um *label* (rótulo, em português) que a identifica permitindo que ela seja alvo de desvios dentro do código.

Laços de repetição e comandos condicionais da linguagem fonte são traduzidos para a representação intermediária através do uso de instruções de desvio para

implementação do seu fluxo de controle. A Figura 8 mostra as duas formas de traduzir o lado direito da produção descrita no topo para o código intermediário C3E. A semântica do fluxo de controle é implementada através dos desvios.

Figura 8 – Leiaute de código para o comando if em C3E



Fonte: Próprio autor

Ao traduzir um código fonte para C3E as chamadas de funções são desmembradas de forma a separar a passagem de parâmetros do desvio para o rótulo inicial da função. A passagem de parâmetros é realizada anteriormente. Na Figura 9 é demonstrado como uma chamada de função dentro de uma atribuição em C pode ser traduzida para C3E.

Figura 9 – Conversão de chamada de função em uma atribuição em C para C3E

```
x = func( vet[i] );     =====➔     1) t1 = i * 4
                                   2) t2 = vet [ t1 ]
                                   3) param t2
                                   4) t3 = call func, 1
                                   5) x = t3
```

Fonte: Próprio autor

As duas primeiras linhas definem a posição do vetor que guarda o valor que será atribuído para a variável temporária t2, possibilitando a sua passagem para parametrização da função na linha 3. Observe-se que na primeira linha o índice definido por i é multiplicado pela constante que representa o tamanho designado para o tipo de dados do vetor. Nesse caso, o exemplo da Figura 9 usa um vetor de inteiros.

A linha 4 faz o desvio para a função informando que apenas um parâmetro foi passado após a linha 3 ter definido qual seria esse parâmetro. Implementações do

C3E podem ser desenvolvidas de forma a dispensar a informação de quantos parâmetros foram passados. Um interpretador poderia simplesmente considerar todos os parâmetros informados desde a última chamada como todos os parâmetros da função chamada.

O C3E também suporta atribuições de endereço e apontador, tais como:

x = &y, onde x recebe o endereço de y;

x =*y, onde y é presumidamente um apontador e a atribuição repassa o valor contido no endereço para o qual ele está apontando para x;

***x = y**, onde o valor contido no endereço apontado por x é alterado para y (AHO, ULLMAN, & SETHI, 2008).

2.2 GERENCIAMENTO DE MEMÓRIA

A Memória Principal, também conhecida como RAM (*Random Access Memory*, ou Memória de Acesso Aleatório, em português) é um importante recurso que deve ser cuidadosamente gerenciado. Por essa razão o SO, no seu papel de gerenciamento de hardware e criador de abstrações úteis, possui uma abstração útil e gerenciável para a memória (TANENBAUM, 2007).

A memória é um recurso escasso, principalmente se a aplicação em questão se ressentir de baixa performance, pois as memórias mais rápidas também são as mais caras e, incidentemente, são voláteis. Os discos rígidos, e as fitas magnéticas antes deles, fornecem capacidade maciça e barata de armazenamento para a grande quantidade de programas e dados que devem ser mantidos. Mas também são muito lentos, sendo que a taxa transferência de dados de discos pode ser até seis ordens de magnitude maior do que a memória principal, e além disso os armazenamentos secundários não podem ser diretamente acessados pelo processador. Para que um programa seja executado as suas instruções devem estar armazenadas na memória principal (Deitel, Deitel, & Choffnes, 2005).

A escolha viável nesse caso, do ponto de vista de *hardware* e SO é utilizar o conceito de hierarquia de memória, reunindo todos os diferentes tipos de memória disponíveis no computador, em geral, *cache*, RAM e disco rígido, em uma única abstração mais simples e útil para os processos que o SO deve servir. Ela contém todos esses diferentes níveis, cada um deles caracterizado distintamente por custo e velocidade.

A parte do sistema operacional que gerencia a hierarquia de memória se chama Gerenciador de Memória e o seu trabalho é gerenciar a memória de forma eficiente: saber quais partes da memória estão em uso, alocar memória para processos quando eles necessitam e “desalocar” memória quando os processos finalizam (TANENBAUM, 2007).

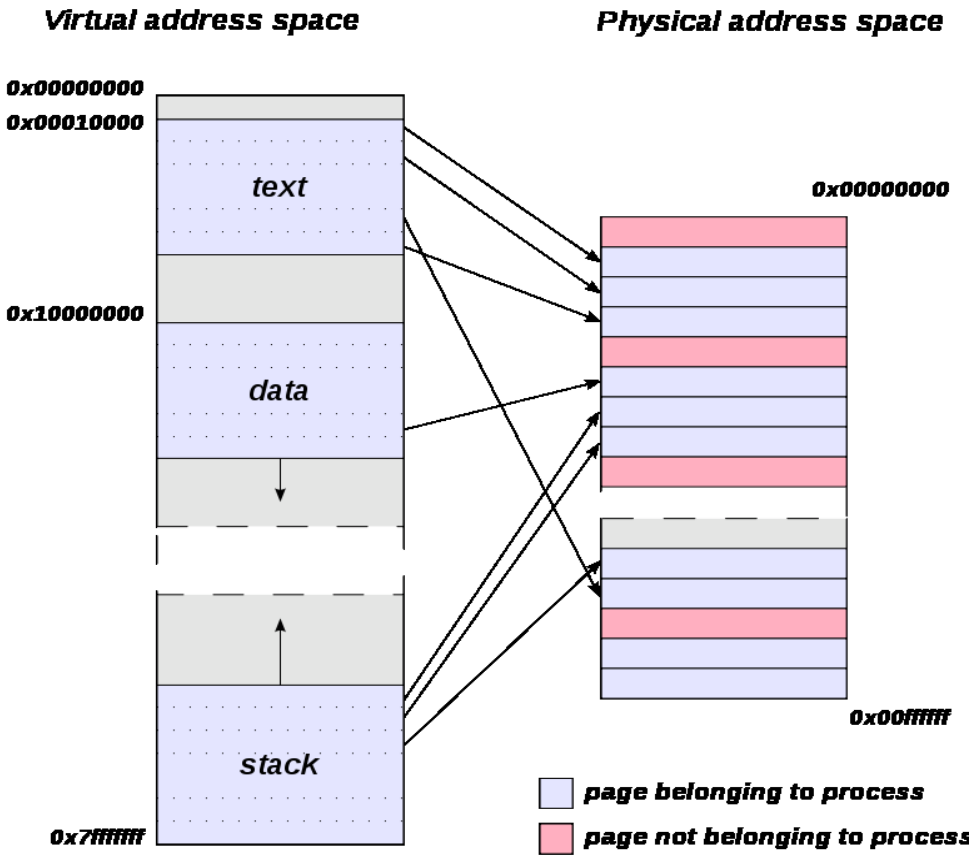
Durante a evolução da arquitetura de computadores e dos sistemas operacionais foram criados diversos esquemas de gerenciamento de memória, desde os muito simples até os altamente sofisticados. Uma das necessidades que naturalmente apareceram foi não expor toda a memória do sistema diretamente aos processos. A intervenção do SO é necessária para a segurança do sistema, impedindo que um processo faça o mapeamento de uma área de memória utilizada pelo SO e o torne inoperável, e também se torna mandatória para o uso da multiprogramação, ou seja, para possibilitar a execução de diversos programas simultaneamente.

2.2.1 *Address Spaces* – Espaços de Memória

Dois problemas precisam ser resolvidos para permitir que duas aplicações estejam em memória ao mesmo tempo: proteção e realocação. Assim como o conceito de processos cria uma espécie de CPU abstrata para cada um deles, a abstração de *address spaces* (espaços de endereços, em português) funciona de forma análoga para a memória, fazendo com que cada processo (inclusive instâncias diferentes de um mesmo programa) viva dentro do seu espaço. Trata-se de uma lista de endereços que o programa pode usar para endereçar memória, independente da lista utilizada por outros processos, exceto em circunstâncias especiais em que dois ou mais processos querem compartilhar o seu *address space* (TANENBAUM, 2007, p. 180).

Isso cria uma diferenciação entre espaços de memória virtual (*virtual address space*) e a memória real, ou memória física (*physical address space*), significando que o endereço de memória 28 codificado (*hard coded*) no código objeto de um programa, na linguagem *Assembly*, por exemplo, será traduzido em endereços físicos diferentes para dois ou mais diferentes processos que estejam executando diferentes instâncias do mesmo programa, ou mesmo de programas diferentes que façam uso do mesmo endereço de memória.

Figura 10 – Esquema de mapeamento de memória virtual para memória física



Fonte:https://upload.wikimedia.org/wikipedia/commons/3/32/Virtual_address_space_and_physical_address_space_relationship.svg

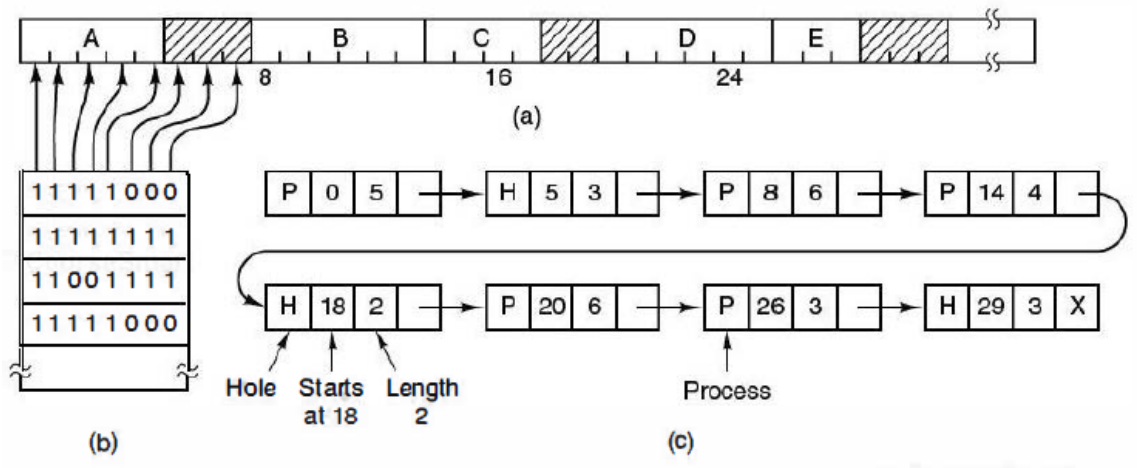
Embora o conceito de *address space* e a sua consequência direta, *virtual memory* (memória virtual, em português) possa garantir a segurança da memória usada pelo SO contra erros de programação, bem como proporcionar condições para o uso da multiprogramação, de forma que um processo não possa mapear memória utilizada por outro, ele não garante que o sistema será capaz de suportar uma situação em que a soma da memória utilizada por todos os processos em execução mais o que está em uso pelo SO for maior do que o espaço físico total da memória RAM.

> Two general approaches to dealing with memory overload have been developed over the years. The simplest strategy, called swapping, consists of bringing in each process in its entirety, running it for a while, then putting it back on the disk. Idle processes are mostly stored on disk, so they do not take up any memory when they are not running (although some of them wake up periodically to do their work, then go to sleep again). The other strategy, called virtual memory, allows programs to run even when they are only partially in main memory.
> (TANENBAUM, 2007)

2.2.2 Gerenciamento de Memória Livre

Em sistemas com arquitetura que suporte multiprogramação e uso de memória virtual, os processos podem não apenas coexistir em memória como também podem sair para o disco e voltar para a memória principal diversas vezes. Além disso todos os processos em execução ainda podem fazer requisições de alocação dinâmica. Isso cria uma questão sobre como gerenciar quais espaços estão tomados ou vazios na memória. Há duas formas de acompanhar o uso de memória: *bitmaps* (mapas de bits) ou então *free lists* (listas livres).

Figura 11 – Esquema de mapeamento de memória com Bitmaps e Free Lists



Fonte: (TANENBAUM, 2007, p. 185)

2.2.2.1 Gerenciamento com Bitmaps

Nesse caso a memória é dividida em unidades de alocação cujo tamanho varia entre algumas *words* (cada *word* equivale a quatro *bytes* em processadores de 32 bits, ou arquitetura x86) e vários KBs. Para cada uma dessas unidades de alocação existe um *bit* equivalente no *bitmap* cujo único propósito é indicar se aquela unidade está ocupada ou não. Um dos possíveis valores binários, zero ou um, indica que a unidade está ocupada enquanto o outro indica que está livre (TANENBAUM, 2007).

A Figura 11(a) representa um trecho de memória que contém cinco processos, diferenciados pelas letras entre A e E, e mais alguns espaços vazios indicados pelas regiões riscadas. As marcas na parte inferior da figura representam as unidades de

alocação. A Figura 11(b) mostra o *bitmap* equivalente para as unidades indicadas pelas setas.

O tamanho da unidade de alocação é uma questão fundamental. Quanto menores forem as unidades de alocação maior o *bitmap* terá de ser e vice-versa. Mapas muito grandes ocupam muito espaço em memória e tornam operações de busca mais onerosas, porém unidades de alocação muito grandes provavelmente causarão problemas para a alocação eficiente de memória, pois a tendência é que espaço seja desperdiçado na última unidade alocada para um determinado processo, a não ser que o seu tamanho em memória seja um múltiplo exato do tamanho determinado para as unidades de alocação.

Tomar uma decisão de utilizar um *bitmap* para o mapeamento de memória pode ter implicações na performance das alocações. Se um processo equivalente a k unidades de alocação será carregado em memória será necessário buscar uma sucessão de k unidades vagas no mapa:

> A bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and the size of the allocation unit. The main problem is that when it has been decided to bring a k unit process into memory, the memory manager must search the bitmap to find a run of k consecutive 0 bits in the map. Searching a bitmap for a run of a given length is a slow operation (because the run may straddle word boundaries in the map); this is an argument against bitmaps.
> (TANENBAUM, 2007)

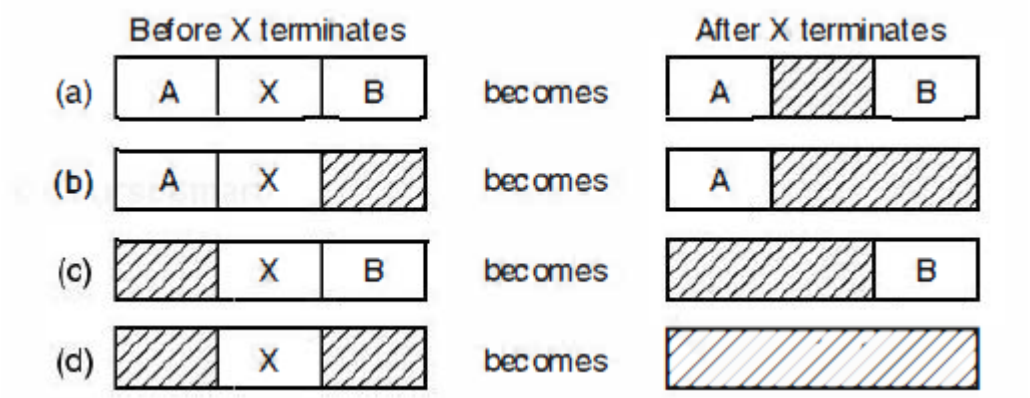
2.2.2.2 Gerenciamento com Listas Livres

Outra forma de manter um registro sobre a disponibilidade da memória é através de uma lista encadeada em que cada um dos nodos representa um segmento de memória e contém o registro de um processo ou de um espaço livre em memória. Cada nodo guarda o endereço inicial e final do segmento de memória que ele representa. A Figura 11(c) apresentada no início da Seção 2.2.4 é uma *free list* que representa o trecho de memória em 11(a) (TANENBAUM, 2007). Cada nodo da lista especifica um buraco livre na memória (H – *hole*) ou um espaço ocupado por um processo (P – *process*) e possui um ponteiro que aponta para o nodo que representa o próximo segmento de memória.

Neste exemplo a lista de segmentos é mantida ordenada por endereço, o que proporciona a vantagem de que quando um processo é terminado ou sofre um *swap out* a atualização da lista é um processo trivial. O nodo de um processo pode ter até

dois vizinhos, ou no mínimo apenas um, caso ele esteja localizado no início ou no fim da lista. A Figura 12 mostra os quatro possíveis cenários de atualização de uma *free list* no caso de um processo finalizado ou que sofre *swap out*.

Figura 12 – Atualização da Free List ao finalizar um processo



Fonte: (TANENBAUM, 2007, p. 186)

Se o processo estava alocado entre outros dois processos, como em 12(a), então o nodo que o representava só tem de ser atualizado para o tipo H, pois agora ele indica um buraco livre na memória entre os processos A e B. No caso de ele estar adjacente a um buraco já existente, de um lado ou de outro, como em 12(b) ou 12(c), o seu nodo deve ser removido da lista (e descartado) e o nodo que representa o buraco adjacente deve ser atualizado de forma que o intervalo de endereços passe a englobar o novo buraco de memória.

E no último caso, ilustrado em 12(d), três nodos da lista devem se tornar apenas um, pois o processo X era adjacente a dois buracos. Depois que ele for finalizado todo o trecho de memória analisado se torna um único buraco. É importante observar que todas essas operações de atualização da estrutura se tornam simples se a lista representando a memória for duplamente encadeada, isto é, se cada nodo apontar também para o seu antecessor, bem como o seu sucessor. Isso facilita muito a navegação para realizar a fusão de nodos.

Organizar os nodos em uma lista encadeada permite o uso de diversos tipos de algoritmos, ou estratégias de posicionamento, para alocação de memória a partir da requisição de um processo. Esses algoritmos determinam em que lugar na memória o sistema deve colocar os dados ou programas que chegam de requisições realizadas por processos (Deitel, Deitel, & Choffnes, 2005).

O mais simples de todos é o chamado *first fit* (primeiro encaixe, em português), que consiste em estacionar o processo no primeiro espaço de memória disponível grande o suficiente para contê-lo, não importando quanto espaço pode sobrar. O nodo representando o buraco na lista será dividido em dois, um para o novo processo e outro para o buraco que sobrou, exceto no improvável caso de que o buraco seja exatamente do tamanho desejado, em cujo caso é só atualizar o nodo para o tipo H. Encontrar um buraco do tamanho exato é a solução ótima para o gerenciamento de memória livre, com o mínimo possível de desperdício de memória.

Uma pequena variação do *first fit* é o chamado *next fit* (próximo encaixe, em português). A sua única diferença é que ao invés de iniciar a busca por um buraco grande o suficiente a partir do início da lista ele busca o próximo *fit* (encaixe) do ponto onde ele parou na sua última leitura da lista. Simulações realizadas indicam que a performance do *next fit* é ligeiramente pior em comparação com o *first fit* (TANENBAUM, 2007, p. 186 apud Bays, 1977).

Outra opção é o *best fit* (melhor encaixe, em português). Este varre a lista inteira em todas as consultas procurando um buraco que seja maior ou igual ao necessário. Caso encontre outro mais adiante que seja menor mas ainda grande o suficiente para acomodar o processo que requisitou a alocação, ele assume esse novo buraco como o melhor encaixe.

Apesar do nome sugestivo esse método é pior do que o *first fit* em questão de performance, pois necessita percorrer a lista inteira para todas as alocações, mesmo tendo encontrado espaços adequados. E além disso também é pior em questão de eficiência no gerenciamento da memória, pois tende a deixar espaços muito pequenos em memória que geralmente são inúteis para novas alocações. Em média, o *first fit* gera buracos maiores.

Para contornar o problema de quebrar buracos previamente existentes em um espaço para o processo e outro pequeno espaço inútil poderia existir um *worst fit*, no qual o maior buraco existente sempre seria escolhido, mas simulações provam que ele também não é uma boa opção. Mas uma boa forma de obter melhoria de performance nas alocações, para qualquer um dos algoritmos previamente apresentados, é manter duas listas separadas para processos e buracos, dessa forma o algoritmo vai se focar apenas em analisar os buracos, sem ter que passar por todos os processos. Entretanto, essa estrutura vai onerar as “desalocações” de memória

pois os buracos criados nos processos terão de ser reinseridos ordenadamente na lista de buracos.

Outra possível otimização seria ordenar a lista de buracos por tamanho, em ordem ascendente, e não de acordo com os endereços. Isso faz com que o *best fit* se torne tão rápido quanto o *first fit* nas alocações. E também elimina qualquer razão para o uso do *next fit*.

E já que as estruturas de controle já foram separadas é possível eliminar por completo a lista de buracos, mantendo as informações da mesma dentro dos próprios buracos. Com isso a primeira *word* do buraco informa o tamanho do mesmo enquanto a segunda aponta para o próximo buraco na memória.

Há ainda uma última opção de algoritmo de gerenciamento denominado *quick fit*, que se baseia em diversas listas contendo buracos de tamanho iguais ou semelhantes com o objetivo de fornecer rapidamente buracos que possuem os tamanhos mais procurados. Isto pode tornar a busca por um determinado muito mais rápida, mas ele ainda assim possui o mesmo problema que outros algoritmos que utilizam listas ordenadas por tamanho, que é a onerosidade da busca pelo local correto para a fusão quando o processo é “desalocado”. Essa realidade pode até mesmo ser agravada dependendo da especificidade do critério usado para separar as listas, pois há processamento envolvido para escolher a lista correta também.

É importante ressaltar que a fusão dos buracos seja sempre realizada:

> If merging is not done, memory will quickly fragment into a large number of small holes into which no processes fit.
> (TANENBAUM, 2007)

2.3 LINGUAGEM C

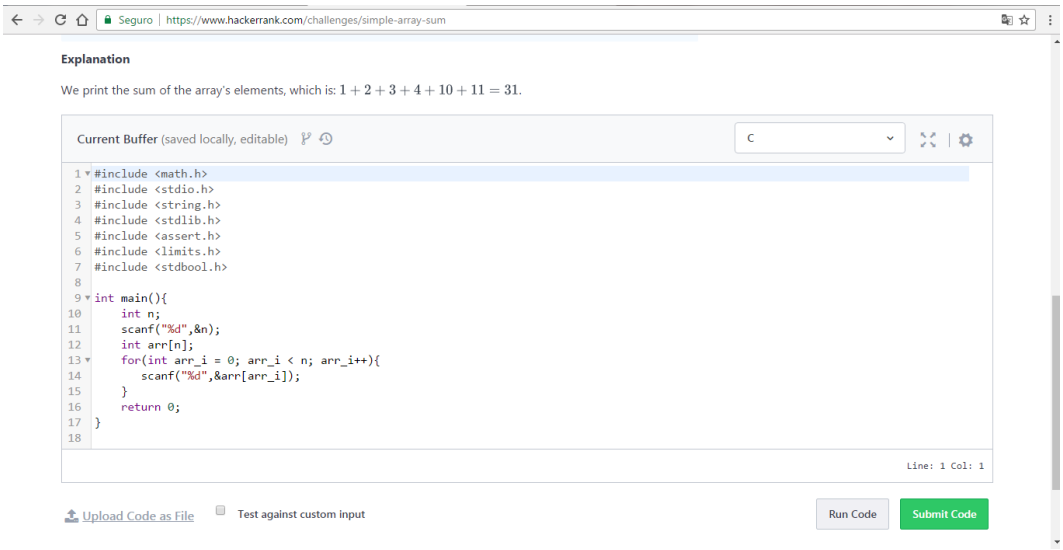
O C é um exemplo de uma linguagem compilada. Existem diversos compiladores disponíveis que convertem código-fonte escrito em C para arquivos executáveis pelo sistema operacional. Nenhum arquivo com a extensão “.c” contendo o fonte do programa pode ser executado diretamente, a não ser que ele o seja por um interpretador da linguagem C.

2.3.1 Simuladores e compiladores online

Existem diversos sites na internet que fornecem funcionalidades semelhantes às de um interpretador em C, mas que na realidade só redirecionam o código informado na página pelo usuário para alimentar um compilador executado no servidor que está do outro lado das requisições de rede. Alguns exemplos destes, de aplicação educacional ou mesmo generalista são: *UVA Online Judge* (https://uva.onlinejudge.org/), *Hacker Rank* (https://www.hackerrank.com/) e o compilador online de C do site *Tutorials Point* (*https://www.tutorialspoint.com/compile_c_online.php*).

Além destes também existe outro projeto na *web* que é inclusive mais abrangente e didático e diversos aspectos. Provavelmente foi iniciado como um tutorial da linguagem *python*, de acordo com o nome e a URL. Trata-se do *Python Tutor* (http://pythontutor.com/).

Figura 13 – Interface de edição e interpretação do Hacker Rank



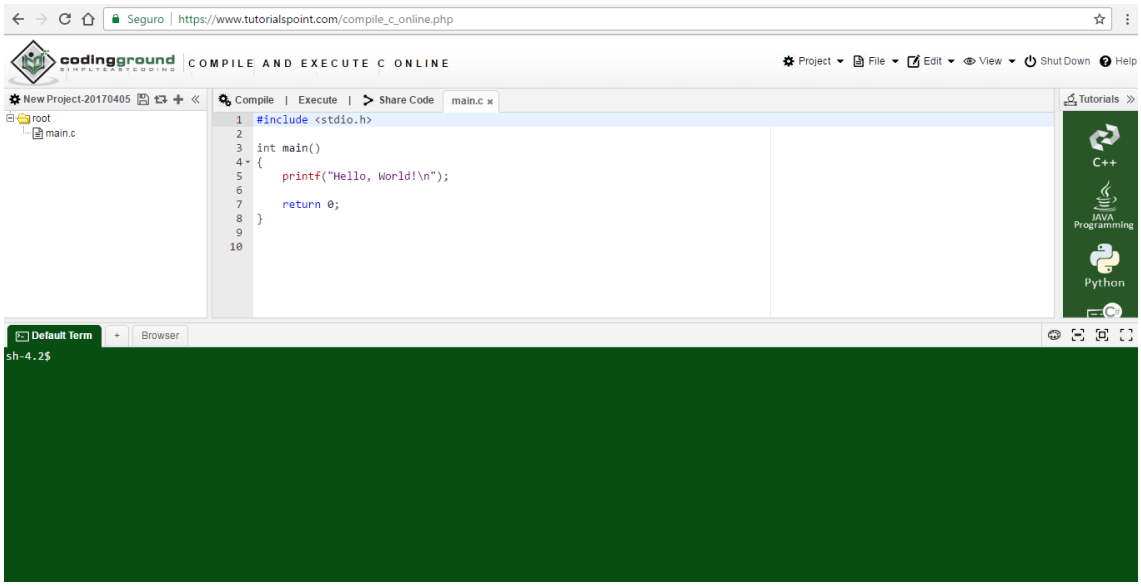
Fonte: https://www.hackerrank.com/

O *Hacker Rank* consiste em uma plataforma não só educativa como também competitiva, permitindo aos usuários competirem com outros programadores no mundo inteiro. Ele se baseia na resolução de diversos problemas de diferentes dificuldades separados em diversas áreas de conhecimento da computação: algoritmos, bancos de dados, estruturas de dados, etc.

Cada problema tem o seu enunciado e apresenta os dados de entrada que serão fornecidos para o algoritmo desenvolvido, quando pertinente, e a saída esperada. Se o programa fornecer a saída correta na simulação realizada após a compilação, então a solução apresentada resolve o problema.

O usuário ainda tem a possibilidade de escrever o seu algoritmo em outras linguagens, se preferir. Problemas de algoritmos, por exemplo, não precisam necessariamente ser resolvidos em C, pois o *Hacker Rank* apresenta um considerável leque de linguagens, entre elas destacam-se:

Figura 14 – Interface de edição e interpretação do Tutorials Point

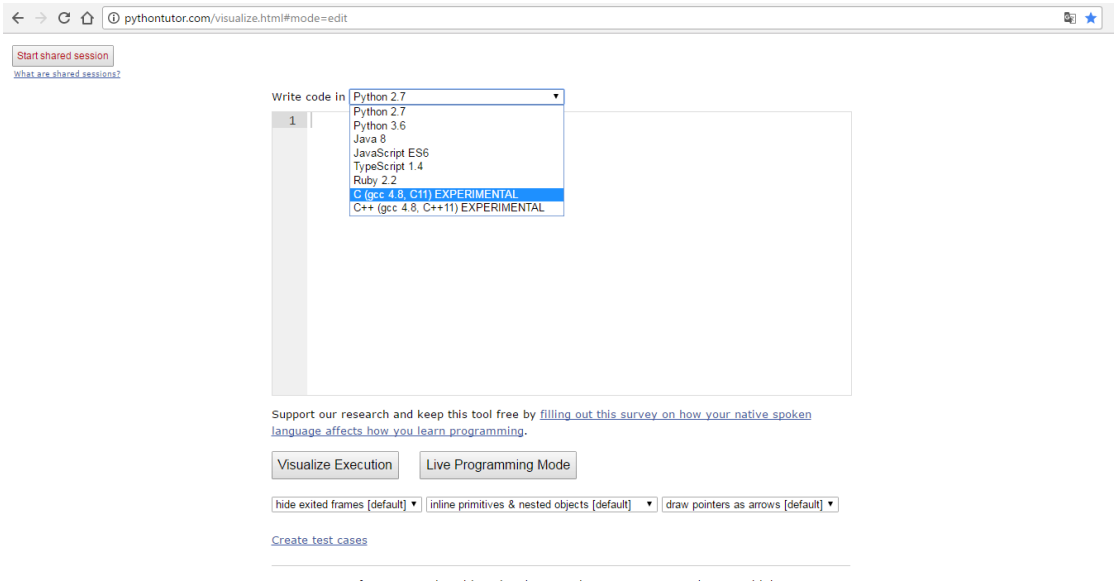


Fonte: https://www.tutorialspoint.com/compile_c_online.php

A ferramenta disponibilizada pelo *Tutorials Point* é basicamente uma IDE livre disponível online. Essa disponibiliza diversas funcionalidades de edição e possui gerenciamento de arquivos em diretórios no canto esquerdo.

O último projeto mencionado nesta seção é o *C* Tutor embutido dentro do projeto iniciado com o nome de *Python Tutor*. Conforme descrito na própria página, apesar do seu nome, ele suporta 7 diferentes linguagens, incluindo C e C++, embora em caráter experimental como evidenciado na Figura 15. Portanto, algumas funcionalidades dessas duas linguagens podem não funcionar por completo no momento do acesso realizado para esta pesquisa.

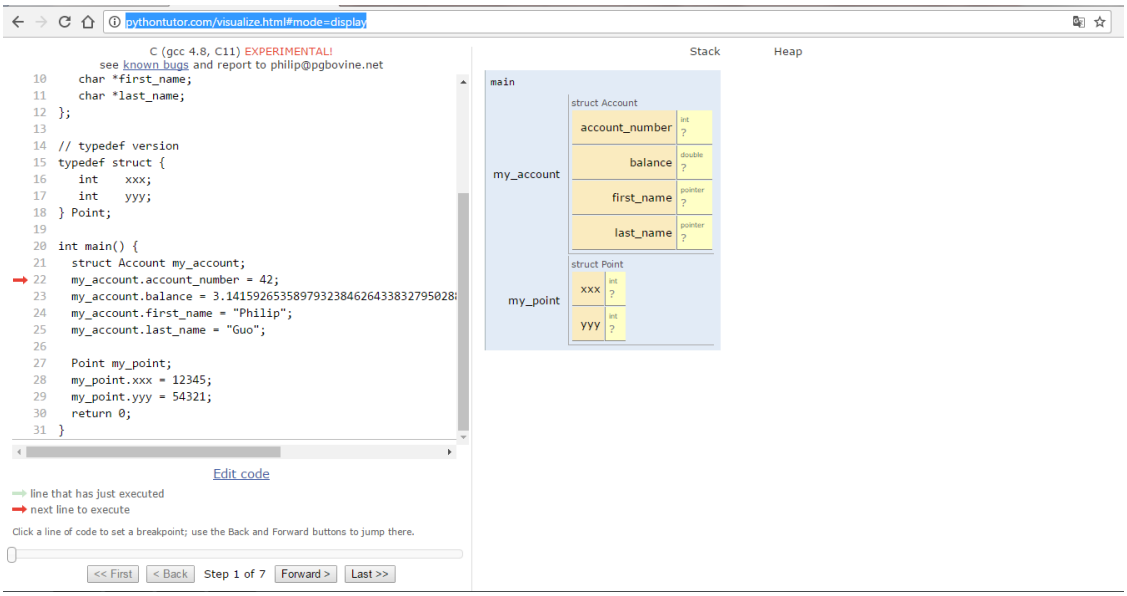
Figura 15 - Interface de edição e interpretação do C Tutor do Python Tutor



Fonte: http://pythontutor.com/c.html#mode=edit

Esta ferramenta possibilita a criação de casos de testes para os algoritmos descritos e apresenta alguns exemplos de programas que demonstram as capacidades embutidas no site, as quais incluem depuração visual conforme demonstrado na Figura 16.

Figura 16 – Depuração visual usando o Python Tutor



Fonte: http://pythontutor.com/visualize.html#mode=display

O WebAlgo, que será expandido para entrar nesse grupo de simuladores, terá funcionalidades similares às de um interpretador pois não vai gerar nenhum programa

executável, como também não o faz para o Português Estruturado, e também será utilizado para educação. Ele apresentará funcionalidades, muitas das quais já implementadas, semelhantes às de IDEs muito utilizadas para edição e compilação de programas em C como o CodeBlocks ou o Dev C/C++.

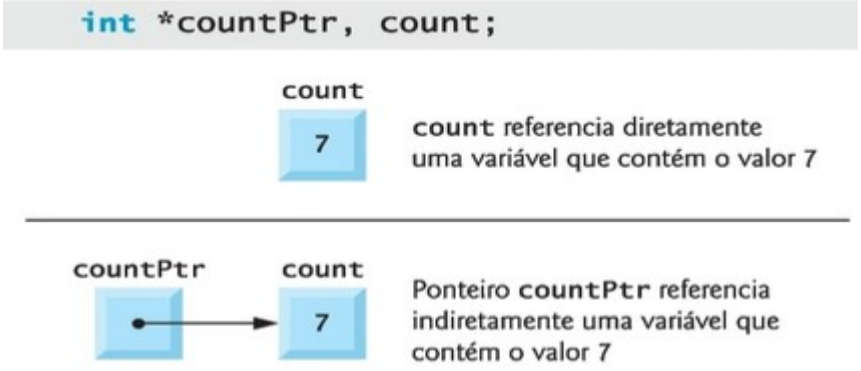
2.3.2 Ponteiros

Ponteiros proporcionam uma forma de trabalhar com referências a endereços de memória, permitindo que os programas escritos em C criem e manipulem estruturas de dados complexas cujo espaço ocupado em memória pode ser variável. Além disso também permite que funções recebam e retornem múltiplos parâmetros por referência.

Portanto, o valor de um ponteiro não é apenas um valor guardado em um endereço de memória, mas o que está no local de memória determinado para o ponteiro é outro endereço de memória que, consequentemente, aponta para outro local na memória. Um nome de variável referencia um valor diretamente, enquanto um ponteiro o referencia indiretamente, de forma que o código objeto gerado por um compilador que fizesse a tradução de um ponteiro não poderia referenciar o endereço de memória diretamente em uma operação, mas sim buscar o endereço que está dentro do endereço original (DEITEL & DEITEL, 2011).

Também é possível codificar um ponteiro que referencia um endereço de memória que contém outro ponteiro, comumente denominado como um ponteiro para um ponteiro. E poderia haver um ponteiro que aponta para este ponteiro indireto, sem haver um limite determinado para as referências indiretas encadeadas.

Figura 17 - Definição de um ponteiro e esquema de referência direta e indireta



Fonte: (DEITEL & DEITEL, 2011, p. 209)

A variável *countPtr* está definida na Figura 17 como um ponteiro para um inteiro. O asterisco ao lado do seu nome indica que *countPtr* é um ponteiro para *int*, ou que ela aponta para uma variável do tipo inteiro. Já a variável *count* é apenas um inteiro convencional, e o esquema na parte inferior da figura mostra a diferença entre os dois.

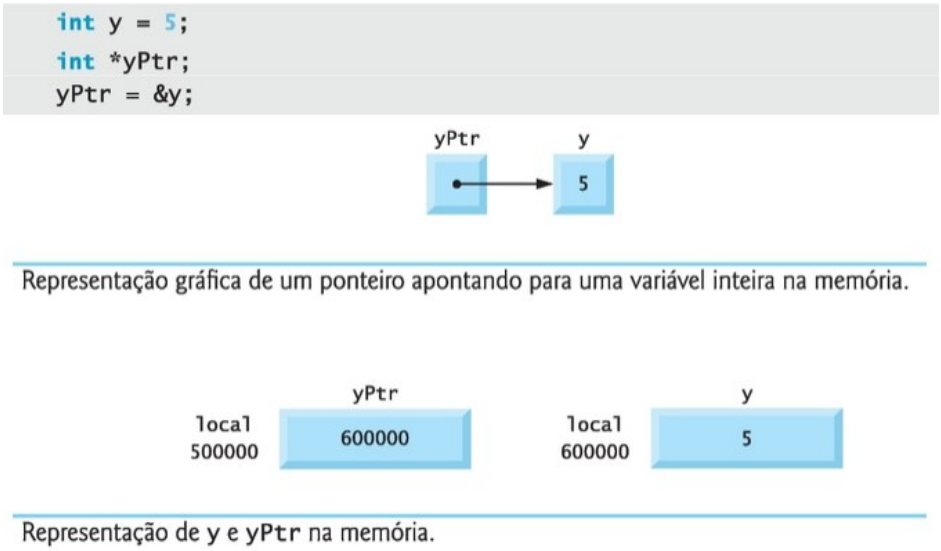
Existem dois operadores unários que permitem a utilização de ponteiros em conjunto com variáveis e outras estruturas comuns. O operador de endereço *&* aplicado sobre uma variável retorna o endereço da mesma em memória, conforme evidenciado na Figura 18.

O outro operador, codificado com o uso de * é normalmente chamado de operador de “indireção” ou “desreferenciação“. Quando usado retorna o valor do objeto apontado pelo seu operando. Portanto, se usado sobre o ponteiro **yPtr**, inicializado na Figura 18, retornaria o valor que está no endereço para o qual ele está apontando, ou seja, o valor de **y**, que nesse caso é 5. Assim, a chamada de função

printf(“%d”, *yPtr);

imprimiria o número 5 em tela.

Figura 18 - Utilização e representação gráfica do operador de endereço



Fonte: (DEITEL & DEITEL, 2011, p. 210)

Após a execução das três instruções no topo da Figura 18 conclui-se que foram declaradas duas variáveis, uma das quais é um ponteiro, a outra foi inicializada com 5 e finalmente o ponteiro recebe o endereço da variável, e não o seu valor, porque a variável **y** foi referenciada com o uso do operador unário de endereço. Abaixo, na

Figura 18, é mostrado o esquema de referência indireta e os valores e endereços das variáveis na memória após a execução do trecho de código especificado.

Os operadores *&* e * são complementares. Se ambos forem aplicados sobre qualquer ponteiro, em qualquer ordem, o resultado produzido será o mesmo: o endereço para o qual o ponteiro está apontando. O uso dos dois operandos ao mesmo tempo faz com que eles se anulem mutuamente (DEITEL & DEITEL, 2011).

Com o uso de ponteiros e operadores de “indireção” em C é possível realizar uma chamada de função simulando uma passagem de parâmetros por referência onde normalmente só é possível fazê-lo por valores. A linguagem C só suporta a passagem de parâmetros por valor. Ao passar um ponteiro ainda assim a função que está chamando passa um valor para a função chamada. A diferença é que esse valor será interpretado como um endereço de memória.

Passando o endereço em memória de uma variável, que pode até mesmo ocupar um espaço muito grande em memória, é possível fazer com que a função possa ler e alterar os seus valores sem ter que copiar todos os seus dados para uma nova estrutura de memória.

Para fazer isso os parâmetros devem ser passados para a função usando o operador de endereço e, na sua declaração, a função deve estar pronta para receber ponteiros como argumentos.

A única exceção a essa regra são os vetores (*arrays*) que são como ponteiros implícitos. Passar para a função apenas o nome de um vetor *arrayName* é o equivalente a *&arrayName[0]*, ou seja, referências ao nome de um vetor resultam no endereço da sua primeira posição.

Os Algoritmos 3 e 4 evidenciam a diferença entre chamar uma função com parâmetros passados por valor e por referência usando ponteiros. Nos dois casos a função é declarada antes de ser codificada e, dessa forma o compilador conhece a sua assinatura, que é a combinação entre o seu nome e os seus parâmetros. Isso permite que o código da função só apareça após a função *main*.

Note-se que no caso do primeiro algoritmo é obrigatório usar o retorno da função para que o programa principal possa obter o resultado da operação. É a única forma de retornar valores de uma função sem o uso de ponteiros, sendo possível retornar apenas um valor. É possível retornar vários campos que estiverem, em última instância, agrupados dentro de uma *struct*, pois é possível ter uma *struct* como o retorno de uma função.

Já no caso da chamada por referência a função pode ser declarada como uma função com tipo de retorno *void*, que é como se permite declarar um procedimento em C, pois o mesmo parâmetro serve para entrada e saída. O que está sendo fornecido para a *cubeByValue* é o endereço de memória da variável. Isso permite que ela faça a alteração diretamente naquele endereço usando o operador de “indireção”.

Algoritmo 3 - Função que calcula o cubo de um número usando chamada por valor

```
1  /* Fig. 7.6: fig07_06.c
2     Cubo de uma variável usando chamada por valor */
3  #include <stdio.h>
4
5  int cubeByValue( int n ); /* protótipo */
6
7  int main( void )
8  {
9     int number = 5; /* inicializa número */
10
11    printf( "O valor original do número é %d", number );
12
13    /* passa número por valor a cubeByValue */
14    number = cubeByValue( number );
15
16    printf( "\nO novo valor do número é %d\n", number );
17    return 0; /* indica conclusão bem-sucedida */
18 } /* fim do main */
19
20 /* calcula e retorna cubo do argumento inteiro */
21 int cubeByValue( int n )
22 {
23    return n * n * n; /* calcula cubo da variável local n e retorna resultado */
24 } /* fim da função cubeByValue */
```

```
O valor original do número é 5
O novo valor do número é 125
```

Fonte: (DEITEL & DEITEL, 2011, pp. 212,213)

Algoritmo 4 - Função que calcula o cubo de um número usando chamada por referência

```
1  /* Fig. 7.7: fig07_07.c
2     Calcula o cubo de uma variável usando chamada por referência com argumento ponteiro */
3
4  #include <stdio.h>
5
6  void cubeByReference( int *nPtr ); /* protótipo */
7
8  int main( void )
9  {
10    int number = 5; /* inicializa número */
11
12    printf( "O valor original do número é %d", number );
13
14    /* passa endereço do número a cubeByReference */
15    cubeByReference( &number );
16
17    printf( "\nO novo valor do número é %d\n", number );
18    return 0; /* indica conclusão bem-sucedida */
19 } /* fim de main */
20
21 /* calcula cubo de *nPtr; modifica variável number em main */
22 void cubeByReference( int *nPtr )
23 {
24    *nPtr = *nPtr * *nPtr * *nPtr; /* cubo de *nPtr */
25 } /* fim da função cubeByReference */
```

```
O valor original do número é 5
O novo valor do número é 125
```

Fonte: (DEITEL & DEITEL, 2011, p. 213)

2.3.3 Structs

Struct é uma abreviação da palavra inglesa *structure*, que significa estrutura. O seu significado na linguagem C basicamente é a junção de diversas variáveis, de diversos tipos e tamanhos, em uma única estrutura heterogênea. As variáveis dentro de uma *struct* geralmente são chamadas de campos ou membros da *struct*.

Após a sua declaração ela pode ser utilizada como um tipo não-primitivo. Todas as *structs* são tipos que consistem de uma sequência de membros cujo armazenamento em memória é alocado em uma sequência ordenada (ISO/IEC, 2011).

As estruturas em C se diferenciam dos vetores pela sua capacidade de armazenar variáveis de diversos tipos simultaneamente. O seu uso em combinação com o uso de ponteiros é que possibilita a formação de estruturas de dados mais complexas, como por exemplo listas interligadas, filas, pilhas e árvores (DEITEL & DEITEL, 2011).

Figura 19 – Exemplo de declaração de uma Struct

```
struct funcionario2 {
   char nome[ 20 ];
   char sobrenome[ 20 ];
   int idade;
   char sexo;
   double salario;
   struct funcionario2 pessoa; /* ERRO */
   struct funcionario2 *ePtr; /* ponteiro */
};
```

Fonte: (DEITEL & DEITEL, 2011, p. 321)

Como o comentário na Figura 19 evidencia, uma *struct* não pode conter uma instância de si mesma, mas ela pode conter um ponteiro para outra *struct* idêntica, o que é chamado de estrutura “autorreferenciada” (DEITEL & DEITEL, 2011). Nesse caso, os registros de funcionários poderiam conter um ponteiro para o seu superior imediato, que também deve ser um funcionário.

Após declarada, a *struct* pode ser usada de forma semelhante a um tipo primitivo, podendo haver diversas instâncias de uma *struct*. A Figura 20 demonstra como declarar uma variável cujo tipo é uma *struct*, como declarar um *array* de instâncias da *struct* em questão e como declarar um ponteiro para uma *struct*. Todas essas declarações podem ser feitas diretamente na declaração da *struct*.

Figura 20 - Instanciando uma struct

```
struct card aCard, deck[ 52 ], *cardPtr;
struct card {
   char *face;
   char *suit;
} aCard, deck[ 52 ], *cardPtr;
```

Fonte: (DEITEL & DEITEL, 2011)

A declaração também pode ocorrer suprimindo o nome do tipo complexo criado implicitamente para a *struct*. A última declaração da Figura 20 poderia ser feita da seguinte forma:

```
struct {
        char *face;
        char *suit;
} aCard, deck[ 52 ], *cardPtr;
```

Entretanto, nesse caso todas as instâncias da *struct* devem ser declaradas na declaração da própria *struct*, ou seja, não será possível declarar no meio do código.

Quanto às operações que podem ser realizadas nas estruturas as únicas válidas são: atribuição de variáveis da estrutura a variáveis de mesmo tipo, coleta de endereço de uma variável da estrutura (a partir do operador de endereço *&*), acesso aos membros de uma instância da *struct*, além do uso da função *sizeof* para determinar o seu tamanho (DEITEL & DEITEL, 2011). E ainda é possível atribuir uma *struct* inteira para outra, se elas tiverem os mesmos campos.

Se for necessário realizar uma comparação entre duas instâncias da mesma *struct*, isso deve ser realizado campo a campo, de preferência dentro de uma função especializada que as receba através de ponteiros. Não é possível simplesmente usar os operadores de comparação usados para tipos numéricos primitivos. Os membros da *struct* não são necessariamente armazenados em *bytes* consecutivos na memória.

A inicialização pode ocorrer a partir de listas de inicializadores, assim como ocorre com vetores, seguindo o nome da variável na declaração com um sinal de igual e uma lista delimitada por chaves. Os valores da inicialização devem ser informados na mesma ordem de declaração dos seus referidos campos receptores, dentro das chaves e separados por vírgulas, como demonstrado abaixo.

```
struct card aCard = { “Três”, “Copas” };

struct carro {
        int id;
        char *nome;
        char *montadora;
} fiesta = { 0, “Fiesta”, “Ford” };
```

Para acessar um dos membros de uma estrutura pode-se utilizar o chamado operador de membro, que consiste em usar um ponto entre o nome da variável declarada como *struct* e o nome do membro. Para referenciar os membros da estrutura que representa um carro declarada acima na função *printf* basta usar o trecho de código abaixo.

```
printf("id: %i\n",fiesta.id);
printf("id: %s\n",fiesta.nome);
printf("id: %s\n",fiesta.montadora);
```

Entretanto, se a instância da *struct* foi referenciada a partir de um ponteiro, como é o caso na maioria das aplicações reais, o uso do ponto não vai trazer o resultado esperado, pois o ponteiro precisa ser "desreferenciado" para possibilitar a busca do campo membro. Para isso utiliza-se o chamado operador de ponteiro de estrutura "->" (DEITEL & DEITEL, 2011). O Algoritmo 5 evidencia a diferença entre uma *struct* declarada normalmente e um ponteiro para uma *struct*.

Algoritmo 5 - Declaração e uso de structs

```c
#include <stdio.h>
#include <string.h>

struct carro {
    int id;
    char *nome;
    char *montadora;
} fiesta = {0, "Fiesta", "Ford"};

int main()
{
    printf("id: %i\n",fiesta.id);
    printf("nome: %s\n",fiesta.nome);
    printf("montadora: %s\n",fiesta.montadora);

    struct carro *outroFiesta;
    outroFiesta = &fiesta;

    printf("id: %i\n",outroFiesta->id);
    printf("nome: %s\n",outroFiesta->nome);
    printf("montadora: %s\n",outroFiesta->montadora);

    return 0;
}
```

Fonte: Próprio autor

Após a atribuição do endereço da *struct* para o ponteiro este passa a apontar para o endereço de memória daquela. Portanto, todos os valores são idênticos à original, inclusive acessando exatamente os mesmos locais em memória.

Utilizar o operador de ponteiro de estrutura é o equivalente a utilizar o operador de “indireção” sobre a *struct* e o operador de membro sobre o campo a ser referenciado, mas necessariamente nessa ordem de precedência, por isso o uso de parênteses no exemplo abaixo:

outroFiesta->id equivale a: **(*outroFiesta).id**

Outra funcionalidade do C frequentemente associada com o uso de *structs* é a criação de sinônimos com a palavra-chave *typedef*. Como o próprio nome sugere ela possibilita a criação de um tipo customizado que, baseado em uma *struct*, permite que instâncias da mesma sejam declaradas da mesma forma com que os tipos primitivos o são. Usando a declaração da *struct* carro feita acima seria possível definir um tipo para a mesma com a seguinte linha de código:

```
typedef struct carro Carro;
```

Isso possibilita a declaração de instâncias da *struct* carro da seguinte forma:

```
Carro fiesta;
```

Ou então o *typedef* poderia ser usado diretamente na definição da *struct*, simplesmente informando o nome do tipo no final, e o mesmo resultado seria obtido:

```
typedef struct {
        int id;
        char *nome;
        char *montadora;
} Carro;
```

2.3.4 Gerenciamento de Memória em C

Gerenciamento de memória é um dos aspectos mais importantes da programação de computadores. Em muitas linguagens de *scripts* e linguagens modernas que usam orientação a objetos, como Java e C#, o programador não

precisa ter qualquer tipo de preocupação com o gerenciamento da memória, que é realizado automaticamente, embora nem sempre de forma eficiente para todos os tipos de aplicações (Bartlett, Inside Memory Management, 2004).

Mas em muitas linguagens, incluindo o C, o gerenciamento de memória é mandatório se o objetivo é trabalhar com qualquer estrutura de tamanho variável em memória. Ao contrário do que ocorre em muitas linguagens orientadas a objetos, onde o gerenciamento de memória é transparente para o programador, permitindo que ele requisite mais memória simplesmente criando novos objetos, em C é necessário controlar quando memória será alocada, realocada e “desalocada”.

Os tipos primitivos e com tamanhos fixos em C também devem ocupar espaço em memória. Estão localizados no que se denomina *stack*. Cada nova *thread*, que é um dos potencialmente vários fluxos de execução dentro de um processo, recebe um espaço de *stack* para guardar o seu segmento de programa (só um por processo, e não por *thread*) e as variáveis que não precisam existir além do seu contexto original.

Uma variável declarada dentro de uma função só existe dentro daquele contexto, portanto o espaço de memória que ela ocupa na *stack* estará livre para outras variáveis assim que a execução da função for finalizada. Mas há muitos casos em que dados precisam transcender os limites do escopo da sua função natal, que é o que necessariamente acontece com memória alocada por requisição manual (Bartlett, Programming from the Ground Up, 2003).

Sempre que o programa está manualmente solicitando memória o espaço necessário não é buscado na *stack*, mas sim na *heap*, que é o espaço destinado para grandes objetos. É boa prática alocar objetos muito grandes na *heap*, mesmo que eles não precisem sobreviver após a finalização do seu escopo, para evitar que a *stack* fique sem espaço disponível, ou seja, para evitar o famigerado erro de *stack overflow*.

A realidade da maior parte das aplicações reais deve lidar com espaços de memória limitados e necessidades bem variadas, o que significa que é necessário o desenvolvimento de um meio de atingir os seguintes requisitos:

- Determinar se existe memória suficiente para acomodar a requisição;
- Encontrar e retornar um trecho disponível de memória de tamanho apropriado;
- Retornar um segmento de memória à lista de páginas disponíveis para possibilitar o seu uso por outras partes do programa ou por outros programas

As bibliotecas que implementam esses requisitos são chamados alocadores (*allocators*) pois são responsáveis por alocar e “desalocar” memória. Quanto mais

dinâmico é um programa mais o gerenciamento da memória se torna um problema e mais significativa se torna a escolha de um alocador adequado (Bartlett, Inside Memory Management, 2004).

2.3.4.1 Alocadores padrão C

Existem algumas funções em C para atender aos requisitos citados anteriormente. Dentre elas, as principais e mais relevantes para a análise desta seção são:

- ***void *malloc(long numbytes)***: aloca um número de bytes, determinado através de parametrização, e retorna um ponteiro para a área de memória reservada, mas sem inicializar a memória. O espaço é simplesmente alocado sem que o seu conteúdo seja alterado. Se não há espaço suficiente retorna um ponteiro nulo, que não aponta para nenhum endereço de memória.

- ***void free(void *firstbyte)***: recebe um ponteiro para um segmento de memória alocado previamente pela função *malloc* e o retorna para uso posterior pelo próprio processo/programa ou retorna diretamente para o sistema operacional. Algumas implementações só podem retornar memória para o programa, e não para o sistema operacional.

Essas duas funções são extensamente utilizadas por desenvolvedores C, mas considerações sobre como elas são implementadas geralmente não fazem parte dos cursos de programação, embora possam ajudar a demonstrar o que está envolvido no gerenciamento de memória (Bartlett, Inside Memory Management, 2004).

Ainda existem outras duas funções para alocação dinâmica de memória oferecidas pela biblioteca de utilitários genéricos *stdlib.h*: *calloc* e *realloc*. Elas podem ser utilizar para criar e modificar estruturas dinâmicas de memória (DEITEL & DEITEL, 2011).

- **void *calloc(size_t nmemb, size_t size)**: aloca uma quantidade de memória igual a *nmemb * size*. O primeiro parâmetro indica o número de blocos de memória e o segundo o tamanho de cada bloco. Assim como a *malloc* retorna um ponteiro para *NULL* se não houver memória suficiente, mas, ao contrário da *malloc*, se houver espaço suficiente para a alocação, a *calloc* inicializa todo o espaço alocado em memória com zeros.

- **void *realloc(void *ptr, size_t size)**: altera o tamanho do bloco de memória referenciado pelo ponteiro *ptr* para *size*. O ponteiro pode referenciar memória previamente alocada pela função *malloc* ou *calloc*. Se o ponteiro apontar para *NULL* antes da chamada dessa função então ela terá um comportamento igual a *malloc*. Se o ponteiro retornado for *NULL* não há memória disponível.

2.3.4.2 Alocadores Alternativos

As funções de alocação alternativas apresentadas em (Bartlett, Inside Memory Management, 2004) foram desenvolvidas com o intuito de substituir as funções padrão do C (ver Apêndice A).

Primeiramente o artigo se baseia em conceitos de memória virtual e funções nativas de mapeamento de sistemas baseados em Unix para apresentar o conceito de *system break*, que é o limite da memória mapeada para o processo.

Sempre que um processo é criado uma certa quantidade de memória é mapeada para ele seja na RAM, ou mesmo em disco caso *swapping* seja necessário. O limite da memória mapeada para o processo se chama *system break*. Quando o processo necessita de mais memória ele deve solicitar mais ao sistema operacional, nesse caso avançando o *system break*, ou seja, fazendo com que o SO mapeie mais memória e retorne um novo *system break*.

Há uma função disponibilizada pelos sistemas Unix denominada *sbrk*. Ela aceita um parâmetro numérico que representa o número de *bytes* solicitados para o SO. Se o parâmetro for zero ela retorna o endereço do *system break* atual.

Ao usar a função *malloc* alternativa o algoritmo verifica se já foi reservado um bloco grande o suficiente para conter a solicitação, não importando o quanto maior ele possa ser. A verificação é realizada através de uma estrutura desenvolvida para ser alocada no início de cada bloco indicando qual o tamanho do mesmo e se ele está ocupado ou não, mas de forma transparente para o programa de aplicação.

Caso não exista um bloco grande o suficiente ocorre uma requisição ao SO para dilatar o *system break*. A função de alocação então marca o bloco como utilizado e retorna-o para utilização. E a função *free* marca o bloco em questão como livre para a utilização de uma futura requisição, sem realizar nenhuma fusão de buracos ou algo equivalente.

Não apenas por esse aspecto, mas também por outros problemas o gerenciador proposto deixa muito a desejar, de acordo com o próprio autor. Entre os problemas mais relevantes estão o fato de que o algoritmo de alocação simplista tende a fragmentar a memória e de que as variáveis utilizadas para controle ocupam um espaço desnecessário, pois o indicador de disponibilidade do bloco só precisaria ocupar um bit de memória. Além disso, não foram implementadas tratativas de exceções, principalmente se o sistema não tiver mais memória disponível (*out-of-memory exception*).

O uso do controle de disponibilidade dentro dos próprios blocos significa que, para encontrar um trecho de memória para alocação o algoritmo vai ter que passar por potencialmente todos os blocos de memória reservados para o processo, muitos dos quais podem estar em disco, o que pode acarretar em muitas operações de *swapping* que terão um impacto negativo sobre o desempenho.

Como esse estudo de caso evidencia, juntamente com os conceitos apresentados na seção sobre gerenciamento de memória, a construção de um gerenciador de memória necessariamente deve priorizar algumas funcionalidades, geralmente relacionadas a performance, em detrimento de outras. As funcionalidades mais importantes são: velocidade de alocação, velocidade de “desalocação”, comportamento para quando a memória está quase cheia, *overhead* de memória para gerenciamento, comportamento em ambientes de memória virtual e tamanho dos objetos alocados.

2.3.5 Structs com alocação dinâmica de memória

É frequente em programação C o desenvolvimento de programas que manejam estruturas de dados complexas cujo tamanho em memória é variável. Isso é proporcionado, conforme demonstrado anteriormente, pelo uso combinado de ponteiros, *structs* e as funções de alocação dinâmica de memória (DEITEL & DEITEL, 2011).

A estrutura “autorreferenciada” abaixo é do tipo que geralmente é utilizada para a construção de listas encadeadas, na qual um nodo da lista aponta para o próximo, através de um ponteiro. Poderia haver um ponteiro para o nodo anterior na lista, em cujo caso seria uma lista duplamente encadeada, que é mais fácil de navegar, porém ocupa mais espaço em memória.

```
struct nodo {
        int dados;
        struct nodo *proximoPtr;
}
```

O restante dos membros da estrutura podem ser qualquer número de campos e de qualquer tipo, não importando para os propósitos de codificação do programa, quanto espaço eles ocupam em memória, pois para definir o número de *bytes* que devem ser solicitados à função *malloc* simplesmente pode-se medir o tamanho de um nodo através da função *sizeof*, conforme código abaixo:

```
novoPtr = (struct nodo *) malloc( sizeof( struct nodo ) );
```

3 ESTRUTURA ATUAL DO WEBALGO

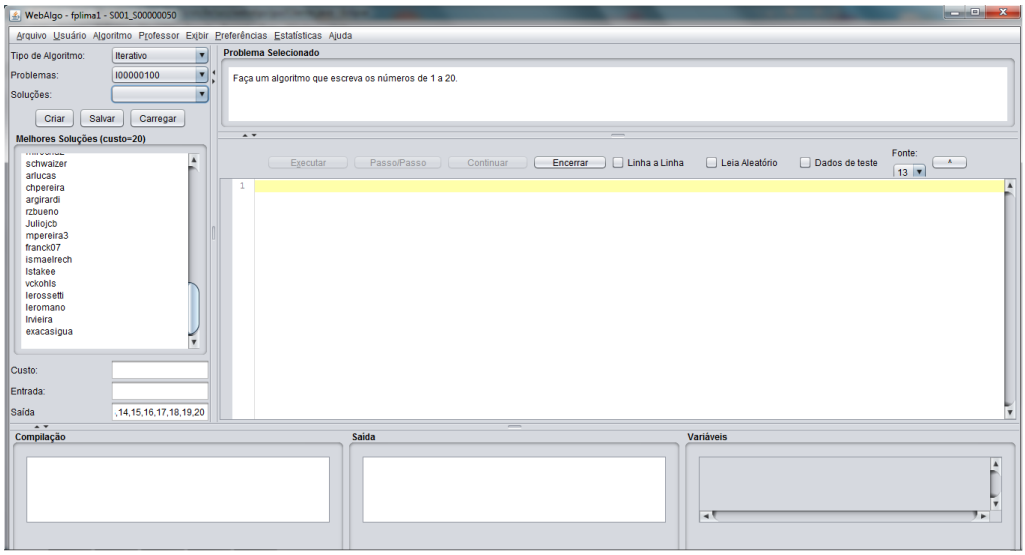
Os tópicos dessa seção foram separados de forma a fornecer uma ampla, porém resumida, visão do sistema educacional de que este projeto trata. Os formalismos e diagramas utilizados para descrever estruturas de classes , objetos e trocas de mensagens estão todos dentro do padrão *Unified Modeling Language* (Linguagem Unificada de Modelagem, na tradução em português), mais conhecido pela sigla UML. Para isso foi utilizada a bibliografia de (Medeiros, 2004).

Os diagramas construídos podem estar desatualizados pois durante o andamento deste projeto o WebAlgo foi, simultaneamente, sofrendo alterações alheias ao que está proposto aqui. Embora as alterações em código possam eventualmente não acarretar em nenhuma alteração nas descrições e diagramas a seguir, é provável que possam se encontrar ocorrências em que um atributo listado aqui não exista mais na versão corrente, ou vice-versa.

3.1 INTERFACE E USABILIDADE

Trata-se de uma ferramenta educacional ampla. A capacidade de reconhecer e compilar o Português Estruturado, o que mais recentemente foi expandido também para a linguagem C, é direcionada para o uso didático. A Figura 21 apresenta a interface do WebAlgo.

Figura 21 – Interface inicial do WebAlgo



Fonte: Próprio Autor

O usuário seleciona um tipo de problema dentro de uma categoria cadastrada na opção em lista localizada no canto superior esquerdo da tela. A lista de problemas logo abaixo carrega os problemas categorizados dentro do tipo de problema escolhido. Ao alterar o problema na lista o seu enunciado é carregado na caixa de texto logo à direita.

Também é possível cadastrar diversas soluções para o mesmo problema. O código da solução será apresentado na área de edição que é a maior caixa de texto localizada no centro. Essa é a área de edição do código fonte, que fornece a funcionalidade de destacamento dos diferentes tipos de construções da linguagem mostrados em tela com diferentes cores, da mesma forma que qualquer outra IDE moderna.

3.2 ANÁLISE LÉXICA

A análise léxica é realizada pela classe **ALexicoC**. Esta foi projetada de forma a retornar o próximo *token* do *buffer* de leitura de acordo com a necessidade da análise sintática. Ao invés de ler todos os *tokens* antes de iniciar a análise sintática e guarda-los em uma estrutura sequencial, estes são lidos um a um durante a análise sintática através da chamada do método *nextToken*.

O analisador léxico também permite voltar ao *token* anterior através dos métodos *setPosition* e *getPosition*. Isso permite ao sintático analisar um ou mais *tokens* à frente e retornar a uma posição anterior antes de invocar um método especializado para a análise de um comando específico.

O *input* de entrada é fornecido ao construtor da classe pelo analisador sintático que por sua vez o recebe como uma única *string* fornecida pela camada de interface. A origem é a propriedade de texto do campo de edição do código fonte.

Cada chamada do método gera um novo objeto da classe **Token** imediatamente guardado como sendo o *token* atual da leitura. Portanto o papel desse método também é o de classificar o que ele está lendo do *buffer* de entrada, e não apenas o de retornar os lexemas, um a um, ao analisador sintático.

A análise léxica foi desenvolvida como um autômato de reconhecimento, optando por um novo estado dependendo do primeiro caractere lido no lexema atual.

Se iniciar por um caractere alfanumérico é uma *string,* o que significa um identificador ou uma palavra reservada. Esta separação fica a cargo do método *trata_string*.

O desenvolvimento da **ALexicoC** foi baseado na sua antecessora e equivalente para o compilador de Português Estruturado, a **ALexico**.

3.3 ANÁLISE SINTÁTICA

A análise sintática é realizada pela classe **ASintaticoC**, que também é responsável por iniciar a execução do interpretador de código intermediário. As mesmas funções já existiam no compilador de Português Estruturado, dentro da classe **ASintatico**. As duas diferem principalmente nas funções de reconhecimento sintático.

Os analisadores sintáticos de ambas as linguagens herdam de uma classe chamada **Constantes** que pretende reunir todas as constantes necessárias, desde tipos de *tokens*, palavras reservadas e parâmetros de sistema. Além disso outras classes também derivam desta. Entretanto, existe polimorfismo implementado entre as duas classes através da interface **IntfSintatico**.

O analisador sintático utiliza diversos objetos auxiliares que tem o objetivo de representar construções da linguagem. Esses objetos criados durante o processo de análise servem tanto para verificações sintáticas como também para interpretação do código intermediário.

As principais classes que descrevem construções úteis estão descritas abaixo.

3.3.1 Classe NodoExp

Descreve um nodo de uma árvore sintática que representa uma expressão aritmética ou lógica. A árvore iniciada a partir de um nodo em particular pode representar toda uma expressão construída a partir do código fonte ou apenas uma sub expressão dentro desta. Pode apontar para até dois nodos filhos, conforme o padrão para representação de expressões e descreve métodos para caminhar dos nodos pais para os filhos, ou de cima para baixo, imaginando a raiz da árvore no topo.

Um objeto desta classe pode representar, além de operadores e variáveis, constantes e chamadas de função. Possui vários construtores com diferentes parametrizações para os diferentes tipos de construções que as suas instâncias

podem representar. Existe um para cada tipo de variável, um para operações, que representam uma sub expressão, e outro para chamadas de função.

Os métodos para resolução das expressões também estão contidos dentro da **NodoExp**. As expressões são resolvidas de forma recursiva através da chamada do método *buscaValor*. A partir da chamada deste a expressão cuja raiz é o nodo atual se auto resolve chamando métodos especializados da própria classe de acordo com o seu tipo (operação, variável, constante, chamada de função).

3.3.2 Classe OTipo

Cada instância desta classe pode ser entendida como um registro dentro de uma tabela de símbolos, que é construída na forma de uma lista encadeada. Cada um dos elementos da lista guarda uma referência para o próximo elemento e outra para o anterior.

Diversos objetos desta classe representam tabelas dentro de contextos específicos, sem a existência de uma estrutura centralizada para todos os símbolos. Em um objeto que descreve uma função por exemplo, há uma instância de **OTipo** para os parâmetros e outra para as variáveis locais.

3.3.3 Classe ConteudoVariaveis

Descreve os objetos que guardam os conteúdos das variáveis, independentemente do seu tipo ou dimensões. A exemplo de outras codificadas na *package* do analisador, também possui vários construtores e atributos especializados que são utilizados de acordo com as suas especificações.

Esta classe foi desenvolvida com a capacidade de suportar *n* instâncias da mesma variável para atender à necessidade gerada pelas chamadas recursivas de funções. Trata-se de uma funcionalidade ativada através da classe **OTipo** que representa todos os aspectos da variável e, portanto, encapsula a **ConteudoVariaveis** e guarda referências para o primeiro e último valor da pilha de instâncias recursiva, enquanto a própria **ConteudoVariaveis** tem a referência para a próxima instância e para a anterior.

3.3.4 Classe OFuncao

A classe OFuncao existe para instanciar e descrever as funções encontradas no código fonte durante o processo de análise sintática. Isso inclui também a função *main*.

A exemplo das outras classes auxiliares descritas acima a **OFuncao** também está equipada para atuar como um nodo em uma lista duplamente encadeada. Todas as funções declaradas no programa são colocadas em uma lista para que possam ser comparadas posteriormente com os identificadores nas chamadas. Se a função não estiver na lista que se inicia a partir da referência inicial contida na **ASintaticoC** então a mesma não foi declarada.

3.4 CÓDIGO INTERMEDIÁRIO E INTERPRETADOR

O WebAlgo da UCS é algo entre um compilador e um interpretador. Tecnicamente, ele realiza ambas as funções em momentos diferentes, pois o algoritmo desenvolvido pelo usuário é convertido em uma lista duplamente encadeada de linhas de código intermediário, que posteriormente será interpretado para refletir a semântica do programa fonte desenvolvido. O encadeamento duplo existe para possibilitar ao interpretador não apenas seguir adiante dentro do código como também retornar às linhas anteriores. A representação intermediária utilizada é o C3E.

Na estrutura atual do programa o mecanismo por trás da interpretação do C3E gerado durante a análise sintática está na classe chamada **Intermediario**. Esta implementa diversos construtores para possibilitar a criação dos diversos tipos de comandos C3E. Atualmente o WebAlgo reconhece nove tipos de estruturas intermediárias, cada uma das quais requer diferentes atributos para o seu funcionamento.

É importante observar que essa implementação deve ser capaz de representar corretamente todas as linguagens fonte reconhecidas pelo WebAlgo. Atualmente existe uma distinção entre um comando de retorno de função e retorno de procedimento exatamente devido à implementação de reconhecimento de Português Estruturado.

Os atributos da classe **Intermediario** também incluem informações necessárias para tratativas de erro, incluindo a linha do programa fonte

correspondente à instrução intermediária representada pelo objeto. Isso possibilita ao WebAlgo apontar para o usuário a linha na qual ocorreu o problema, o que é muito útil para erros de execução como, por exemplo, divisão por zero.

O interpretador de código do WebAlgo é acionado após o fim da fase de análise sintática, caso esta tenha sido completada com sucesso. A partir do método principal do analisador sintático (método *algoritmo*) a *thread* declarada na classe é inicializada e acionada para executar a lógica principal que está no método *run*, sendo que cada objeto **Intermediario** existente no vetor tem o seu método *executa* invocado a partir do método *proximoComando* da classe **ASintaticoC**.

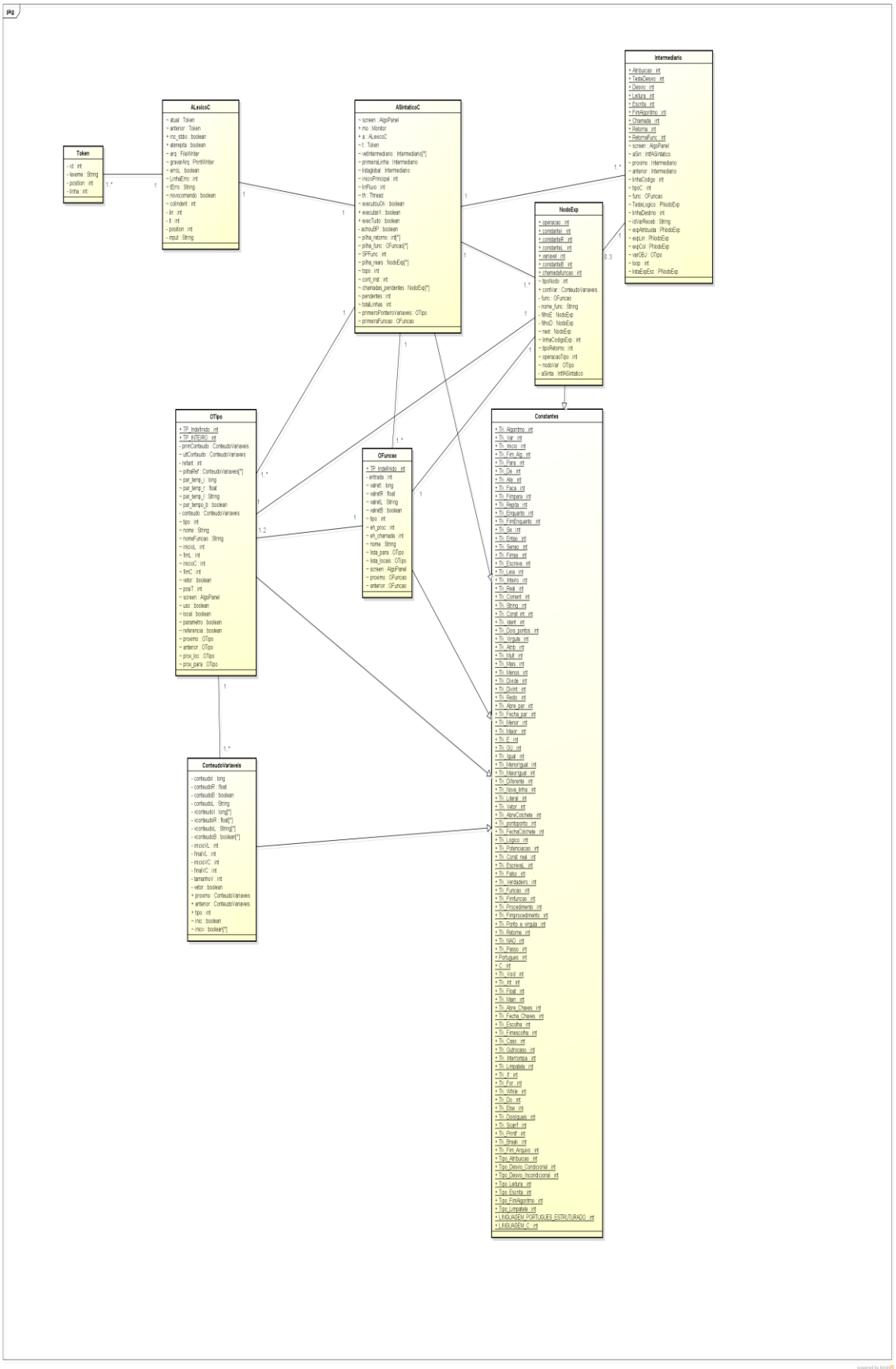
O motivo pelo qual o interpretador de código é iniciado utilizando um fluxo de execução paralelo é para evitar que os elementos da interface fiquem desabilitados enquanto a *thread* principal está aguardando o término da execução do C3E. É necessário que o processo fique rodando em paralelo para possibilitar ao usuário a utilização das funções de depuração e execução passo-a-passo.

3.5 ESTRUTURA DE CLASSES

Na figura 22 está apresentado o Modelo de Domínio completo do compilador/interpretador do WebAlgo, refletindo as classes desenvolvidas dentro da package denominada *br.ucs.webalgo.algo.analisador* dentro do projeto original no *Eclipse*. Este é referente à versão anterior ao projeto.

O Diagrama de Classes completo correspondente está disponível no Apêndice B. O apêndice C apresenta o mesmo diagrama referente ao estado das classes ao final do projeto.

Figura 22 - Modelo de Domínio do Compilador/Interpretador do WebAlgo



Fonte: Próprio autor

4 SOLUÇÕES PROPOSTAS PARA IMPLEMENTAÇÃO

As subseções a seguir foram descritas na forma de um levantamento de requisitos em forma de texto, sem uma separação clara entre formais e não-formais pelo fato de que a maior parte deles se apoia em conceitos desenvolvidos no referencial teórico.

As Seções 2.3.3 até a 2.3.5 servem como requisitos de implementação, enquanto a Seção 2.2 desenvolve muitos conceitos utilizados para o desenvolvimento das estruturas utilizadas.

O texto contido entre as Seções 4.1 e 4.3 descreve a parte central da etapa de projeto deste trabalho, de acordo com o problema de pesquisa e objetivo geral. Na Seção 6 estão descritas as funcionalidades efetivamente implementadas.

Esta seção foi mantida, embora alterada com relação à etapa de projeto, para possibilitar uma comparação entre o projetado e o desenvolvido.

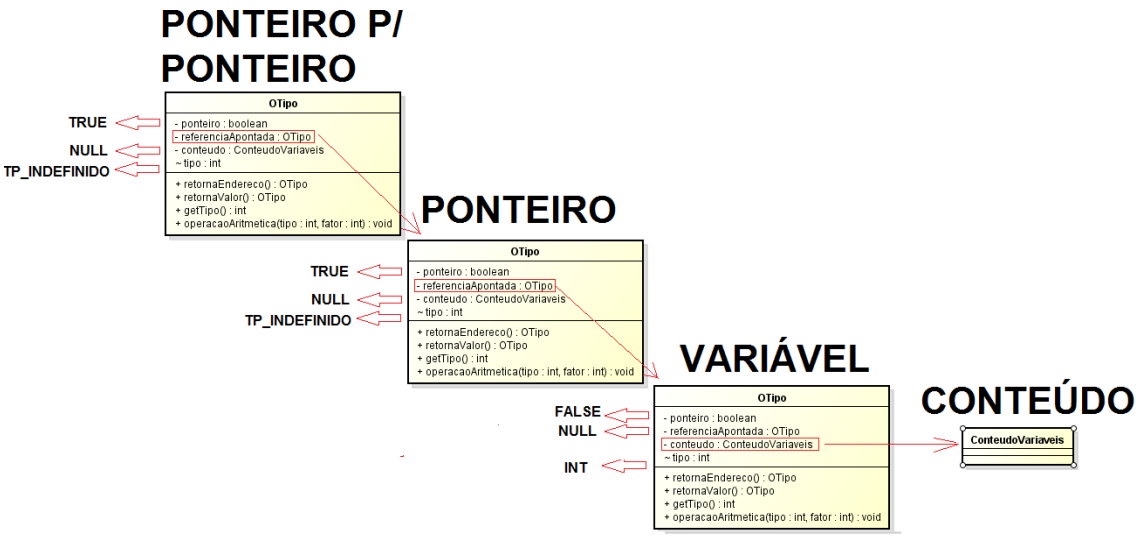
4.1 IMPLEMENTAÇÃO DE PONTEIROS

Gramaticalmente os operadores de endereço e de “indireção” são operadores unários em C, isto é, são aplicáveis sobre apenas um operando, que por sua vez também pode ser o resultado de uma expressão.

Além das alterações necessárias nos analisadores léxico e sintático para possibilitar o reconhecimento do operador de endereço e do operador de “indireção” ainda foi necessário considerar uma estrutura de objetos adequada para suportar a semântica durante a execução do código intermediário. Alterações nas quatro classes auxiliares principais listadas e explanadas na Seção 3.3 se fizeram necessárias. Todas elas implementaram atributos e métodos para possibilitar o gerenciamento de ponteiros.

A classe **OTipo**, quando instanciando um objeto que representa uma variável declarada no código fonte como um ponteiro, não faz uso do seu atributo do tipo **ConteudoVariaveis**. Ao invés disso, guarda uma referência para outra instância da **OTipo** que representa a variável apontada.

Figura 23 - Diagrama de objetos com ponteiros esquematizado



Fonte: Próprio autor

A Figura 23 é um diagrama de objetos modificado que ilustra como funciona o esquema de referências entre objetos na implementação de ponteiros. Este foi desenvolvido ainda durante a etapa de análise e implementado exatamente como foi definido na análise.

Neste caso a referência para a **ConteudoVariaveis** não é utilizada, mas sim uma nova referência para a própria classe **OTipo.** Isso implica em novos atributos e métodos para suportar as novas funções. Alguns métodos tiveram de ser alterados e outros foram criados para atender necessidades semelhantes, pois deve-se contemplar a possibilidade de que o objeto em questão possa ser um ponteiro e verificar para qual que tipo de objeto ele está apontando.

Durante a etapa de projeto foi levantada a necessidade de desenvolver alguma forma de simular a memória *stack*, de forma que cada variável que não fosse alocada na memória *heap* tivesse um endereço para retornar quando fosse utilizado o operador de endereço e, da mesma forma, para resolver quando utilizada a funcionalidade de “indireção”, mas na prática isso foi resolvido apenas retornando as referências de **OTipo** para os ponteiros e com métodos especializados nas classes da estrutura intermediária citadas na Seção 3.3, além de outras estruturas que só foram pensadas na etapa de implementação porque a sua necessidade só surgiu posteriormente.

Um ponteiro pode apontar para uma variável que foi declarada em outro contexto, isto é, em outra função. Observe-se que o contexto em questão pode ainda

estar ativo ou já estar inativo, isto é, com sua execução já finalizada. Se a função *main* chama outra função dentro do seu código enviando para ela um parâmetro que é um ponteiro para uma de suas variáveis locais então a referência é válida. Ela será inválida se, por exemplo, uma função chamada pela *main* retornar para esta, em um ponteiro, o endereço para uma de suas variáveis locais.

Nesse caso, a memória alocada em uma aplicação real em C já poderia ter sido alocada para outro uso. Portanto, isso caracteriza uma prática de programação incorreta, e sendo o WebAlgo uma ferramenta educacional seria desejável que ele emitisse uma mensagem de erro explicando a questão, embora de forma sucinta, para o usuário.

Outra funcionalidade importante que deve ser considerada é a aritmética de ponteiros. Pelo fato de ser um endereço de memória é possível somar e subtrair valores de um ponteiro, ou mesmo fazer qualquer outra operação aritmética sobre ele, fazendo com que ele passe a apontar para um novo endereço. Mas a ação de incrementar um ponteiro de uma unidade, por exemplo, não faz com que ele aponte para o próximo *byte* na memória. Ele vai apontar para o endereço que está a *n bytes* de distância, sendo *n* o tamanho, em *bytes*, do tipo para o qual ele aponta. A Tabela 2 demonstra como funciona o deslocamento de endereços resultantes de operações aritméticas sobre ponteiros para alguns tipos primitivos.

Tabela 2 - Deslocamento de endereços para operações sobre ponteiros

Declaração	Operação	Resultado
int * ptr;	**ptr++;**	Endereço deslocado 4 *bytes* para a frente
int * ptr;	**ptr += 5;**	Endereço deslocado 5 * 4 *bytes* para a frente
char*ptr;	**ptr = ptr - 4;**	Endereço deslocado 4 * 1 *bytes* para trás

Fonte: Próprio autor

O WebAlgo não gerencia memória real, portanto os valores das variáveis não estarão expressos em *bits* para permitir um deslocamento orientado por quantidade

de bytes na memória. Simplesmente haverá um objeto após o outro. E variáveis estáticas não terão um endereço de memória relacionado.

A solução original pensada durante a etapa do projeto é que o suporte à matemática de ponteiros poderia existir para vetores e matrizes já que, de outra forma, usando um compilador convencional, seria uma má prática de programação. Mas a forma como a funcionalidade de vetores e matrizes foi desenvolvida dentro do WebAlgo impossibilitou essa implementação sem que houvesse uma alteração estrutural muito grande.

Vetores dentro do WebAlgo são listas de conteúdos e não de referências, com exceção das novas implementações que contemplam vetores de *structs*. Para que o uso de matemática de ponteiros fosse possível seria necessário realizar complexas conversões que tornariam o código ainda mais complexo. Portanto, essa funcionalidade foi desconsiderada devido a uma ponderação sobre o seu custo-benefício.

Cada elemento de um vetor, por exemplo, teria de ter um indicador apontando para qual posição a sua referência sem o uso de colchetes estaria apontando em um determinado momento e a alteração necessária em diversas classes era grande demais.

A sugestão para o futuro, se essa funcionalidade for desejável, é de que cada variável escalar declarada dentro do WebAlgo esteja representada em um trecho de memória, que seria o equivalente à memória *stack*, como já mencionado anteriormente nesta seção. Além disso vetores e matrizes teriam de ser expressos de forma diferente. Talvez como instâncias de **OTipo** organizadas dentro da própria classe **OTipo** a exemplo do que foi feito para vetores de *structs*.

4.2 IMPLEMENTAÇÃO DE STRUCTS

A implementação necessária para a funcionalidade de *structs* implicou na adaptação da classe **OTipo** para possibilitar a manutenção de uma lista de objetos dessa mesma classe representando os campos da estrutura complexa.

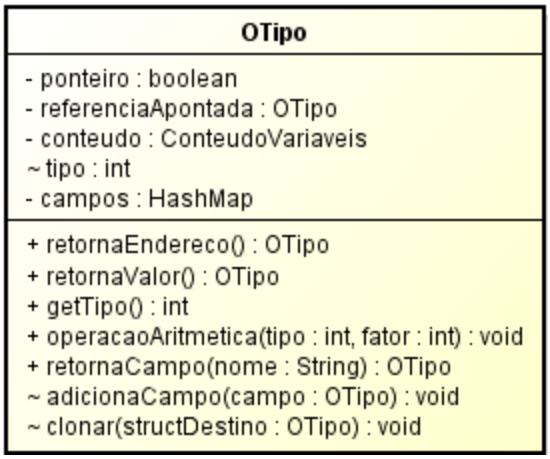
Ao interpretar uma referência para um membro de uma determinada *struct* um novo método dessa classe é acionado para retornar a instância de **OTipo** que representa o campo em questão. Portanto os campos estão classificados de forma que a sua busca esteja indexada pelo seu identificador. Isso foi implementado através

do uso da classe *HashMap* do Java, que permite que objetos sejam indexados através de chaves. Nesse caso, seu índice é o nome do campo. Dessa forma, cada instância de uma *struct* tem uma espécie de tabela de símbolos dedicada.

A Figura 24 apresenta os novos atributos e métodos originalmente previstos para implementação de ponteiros e *structs* na classe **OTipo**. Apenas os três últimos métodos foram de fato utilizados.

Os métodos *retornaEndereco* e *retornaValor* pensados para a interpretação dos operadores de endereço e “indireção” não foram necessários. Simplesmente a referência da variável é diretamente repassada quando necessário. Toda a lógica ficou dentro da classe **NodoExp**. Informações mais detalhadas estão na seção 6.3.

Figura 24 - Alterações previstas na classe OTipo para suporte a structs



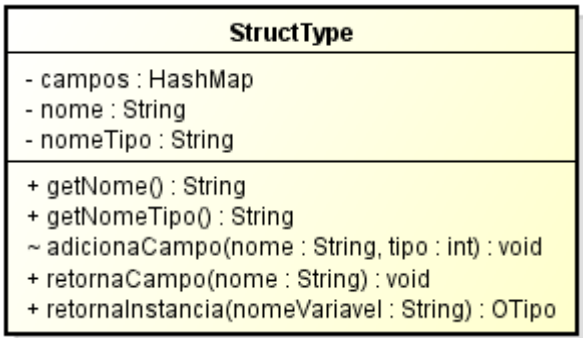
Fonte: Próprio autor

O método *clonar* foi previsto para permitir que os valores de uma *struct* possam ser repassados para outra quando ocorre uma atribuição direta. Para tal é necessário não apenas que ambas contenham membros com identificações idênticas (nome e tipo), mas também que sejam oriundas de um mesmo tipo complexo.

Uma funcionalidade muito importante em C é a possibilidade de construção de tipos customizados complexos através do uso de *structs*. É possível declarar uma *struct* e defini-la como um tipo para uso posterior, permitindo a criação de inúmeras instâncias da mesma *struct*. Para a implementação deste requisito foram criadas novas classes para conter os tipos complexos, pois eles não poderiam ser representados com um objeto do tipo **OTipo**, já que este só deve existir quando a *struct* é, de fato, instanciada. Essa distinção formou uma base muito importante para a implementação de *structs*.

Quando uma variável é declarada com um desses tipos complexos o método *retornaInstancia* retorna um objeto **OTipo** que representa a nova instância. A Figura 25 apresenta a estrutura original da nova classe denominada **StructType**, pensada para atender a essa necessidade.

Figura 25 - Estrutura prevista para suporte a tipos complexos



Fonte: Próprio autor

A principal diferença com relação à estrutura pensada durante a análise para suportar os tipos complexos é que, durante o desenvolvimento, foi verificada a necessidade de uma estrutura separada para definir os campos membros da *struct*, pois existem diversos fatores que se precisam ser considerados mesmo que o campo seja de um tipo primitivo tais como: se ele é ponteiro, ou se é vetor ou matriz.

4.3 GERENCIAMENTO DE MEMÓRIA

4.3.1 Método de dimensionamento de objetos

O primeiro passo foi adotar uma unidade de medida para possibilitar que as estruturas alocadas em memória e também a própria memória simulada sejam mensuradas e controladas. Foi implementado do reconhecimento da função *sizeof* do C. Esta retorna o tamanho de qualquer tipo primitivo, considerando que uma variável pode representar um vetor ou matriz.

Sobre esta questão é importante observar que vetores e matrizes podem não ter tamanho fixo definido. Sabendo que uma *string* em C é declarada como um vetor de caracteres observa-se que a declaração

char * str;

representa uma *string* que pode ter qualquer tamanho. Trata-se simplesmente de um ponteiro para o início de uma *string*.

Esta foi reconhecida durante a primeira etapa do projeto como uma questão notável durante a implementação do simulador de memória, pois o tamanho total ocupado por uma variável com essas características é variável. Neste caso, dependeria de quantos caracteres foram atribuídos à *string str*. Além disso, no momento da alocação da memória a *string* em questão provavelmente estaria vazia.

Vetores de tamanhos variáveis é uma das limitações do WebAlgo, mesmo e inclusive com relação às alterações realizadas neste projeto. O motivo é o mesmo apresentado na seção 4.1 para matemática de ponteiros. No caso da *string* str declarada acima ela é apenas um ponteiro para uma variável do tipo *char*. Qualquer *string* declarada terá que necessariamente ter a sua dimensão pré-determinada.

Pela soma do tamanho de todos os seus membros é que se pode mensurar o tamanho de um objeto do tipo *struct*. Essa implementação também contempla os casos de *structs* aninhadas, pois uma *struct* pode existir dentro de outra.

Portanto, o tamanho de todos os tipos primitivos está expresso em *bytes* codificado em forma de uma constante dentro do WebAlgo. A classe que contém essas constantes está descrita na seção 4.3.2.

4.3.2 Estrutura do Simulador de Gerenciamento

O simulador de gerenciamento de memória desenvolvido nesse projeto segue o modelo de mapeamento de memória com listas encadeadas. Existe uma lista para os espaços em memória disponíveis e outra para os que estão atualmente ocupados.

Para isso foi utilizada a classe *LinkedList*, que é nativa do Java, e implementada como uma lista duplamente encadeada (ORACLE Corporation). Os seus métodos para adição e remoção de membros são mais rápidos do que outras opções disponíveis na linguagem, como o *Vector* e o *ArrayList*, apesar de sua busca ser mais lenta (Lanhellas).

Cada nodo desta lista consiste em um objeto que representa um trecho de memória. Nesse caso trata-se apenas de uma simulação, pois nenhum deles representa um espaço real de memória.

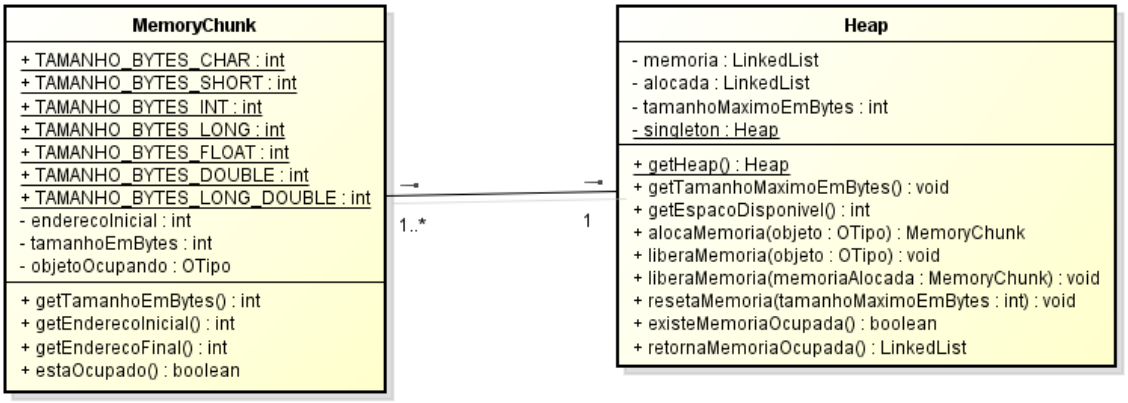
Os nodos possuem atributos e métodos que informam o seu tamanho, endereço inicial e final e um indicador que informa se está ou não ocupado. Além

disso, cada um também guarda uma representação do que ele contém quando ocupado, isto é, um objeto que descreva o conteúdo que ocupa aquele trecho de memória.

E, finalmente, a classe que representa a memória em si descreve um método que informa quanto espaço está disponível em memória para que seja possível verificar a disponibilidade antes de realizar uma alocação.

A Figura 26 mostra um diagrama de classes para estruturação do simulador de memória de acordo com os requisitos descritos acima. Essa é a versão desenvolvida na etapa de projeto

Figura 26 - Estrutura de objetos para gerenciamento de memória

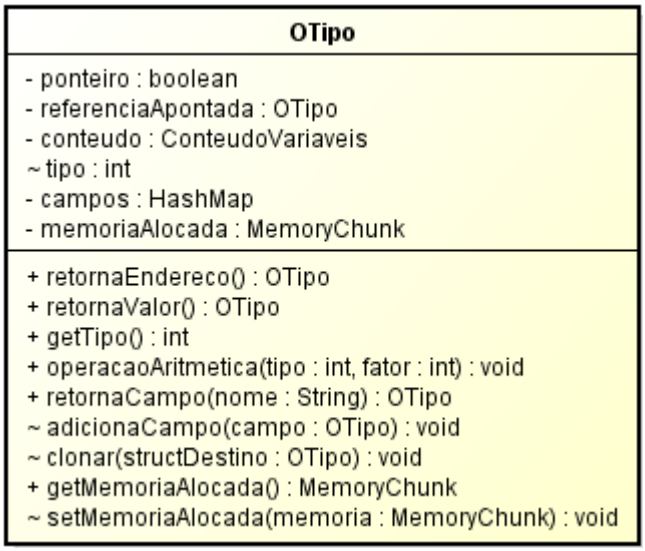


Fonte: Próprio autor

Também foi necessário alterar a classe **OTipo** para que ela passe a suportar uma referência à instância de **MemoryChunk** que representa o espaço de memória no qual o objeto **OTipo** em questão está alocado. A Figura 27 mostra os novos atributos e métodos da classe **OTipo** que, na etapa de projeto, foram definidos como necessários para a implementação do gerenciamento de memória.

Os métodos que lidam com gerenciamento de memória foram de fato desenvolvidos. A principal diferença do projeto para o código desenvolvido é que o trecho de memória ocupado também possui uma referência para o objeto que o está ocupando.

Figura 27 - Alterações na classe OTipo para suporte a gerenciamento de memória

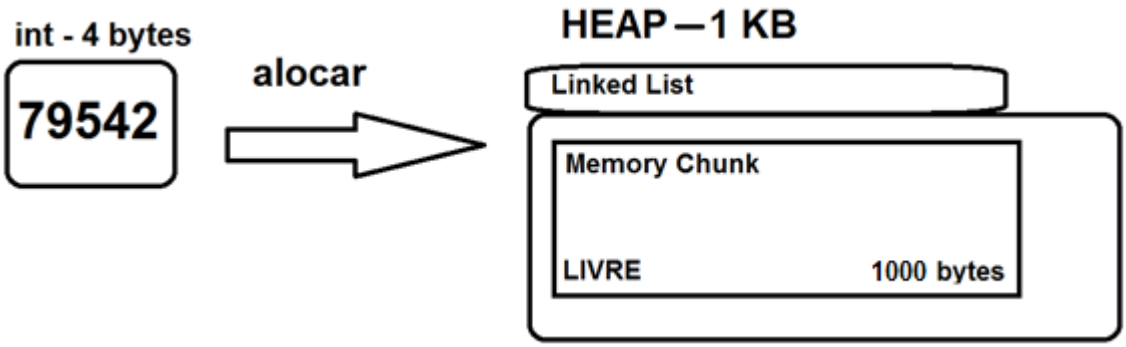


Fonte: Próprio autor

Inicialmente toda a memória disponível na *heap* simulada é representada por um grande buraco que, por sua vez, é representado por um único nodo na lista de espaços desocupados. Foi implementado, conforme previsto, um mecanismo de inicialização do simulador que permite ao analisador sintático reiniciar a memória sempre que um novo algoritmo estiver prestes a ser analisado. Esse mecanismo permite que seja estipulado um tamanho máximo de memória ocupada.

A Figura 28 mostra a memória em seu estado inicial, com um tamanho máximo estipulado de 1 *kilobyte*, no momento em que é realizada uma requisição de alocação para uma variável do tipo inteiro.

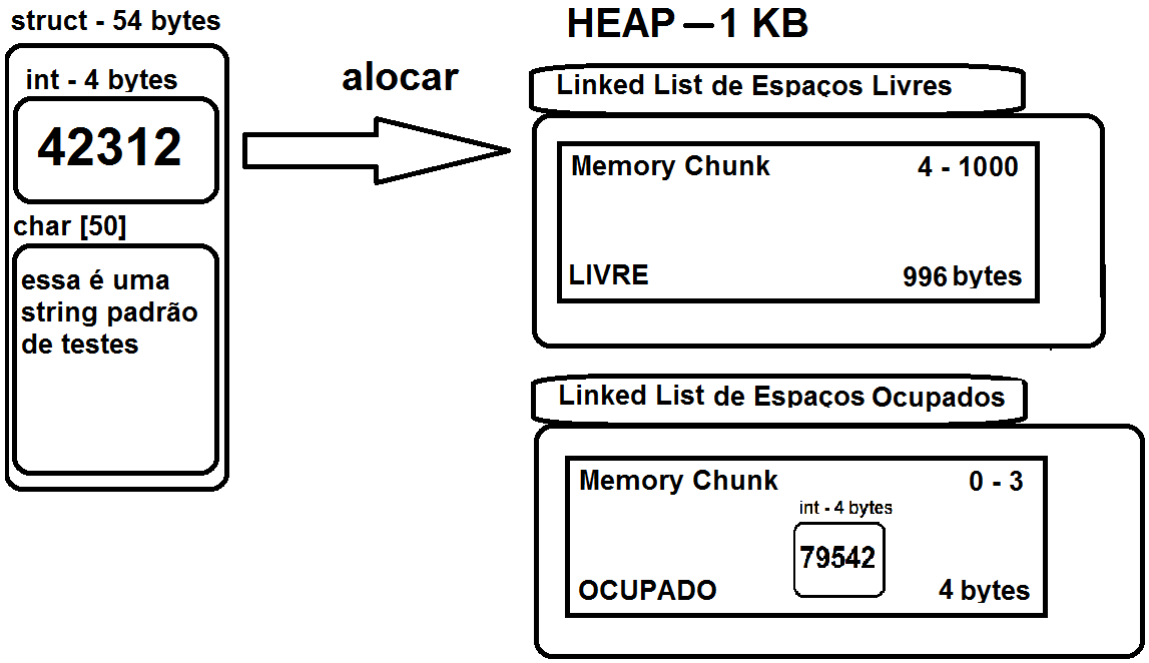
Figura 28 - Memória heap em seu estado inicial



Fonte: Próprio autor

O algoritmo de busca é o *first fit*, o qual aloca a memória necessária no primeiro espaço grande o suficiente para comportar o objeto em questão. Se o espaço escolhido for maior do que o necessário o algoritmo separa a porção ocupada da desocupada e as coloca em listas diferentes que sempre estarão ordenadas pelo endereço inicial para facilitar as novas alocações, ou mesmo “desalocações”, conforme demonstra o exemplo da Figura 29.

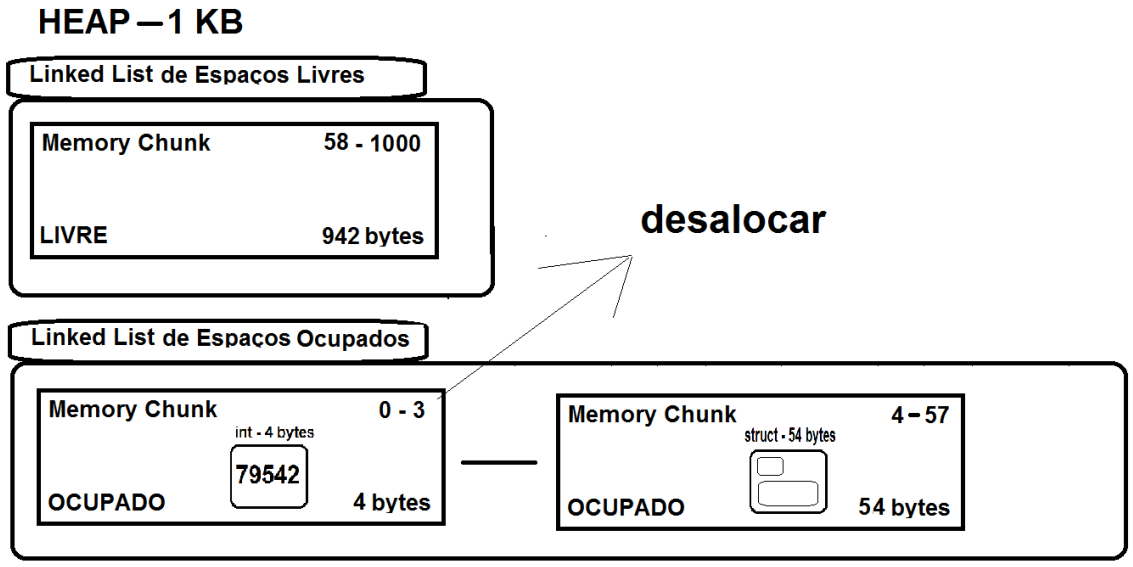
Figura 29 - Listas de espaços de memória separadas após a primeira alocação



Fonte: Próprio autor

Quando ocorre liberação de memória o programa funde o novo espaço disponível a um espaço adjacente na lista de disponíveis, caso de fato exista algum espaço adjacente.

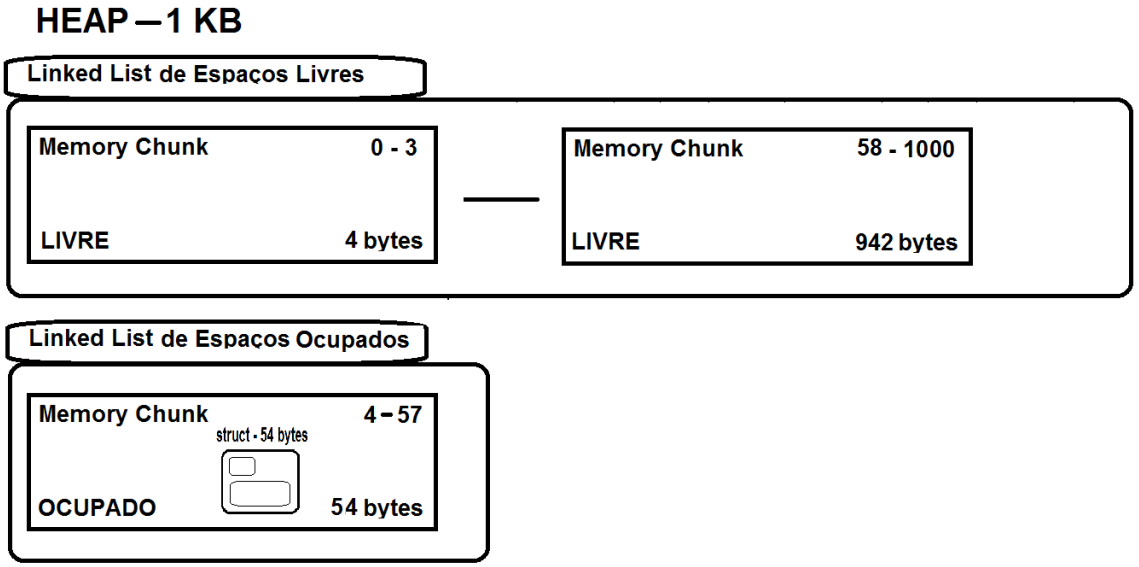
Figura 30 - Estado da memória logo antes de sofrer a primeira liberação



Fonte: Próprio autor

A Figura 30 representa a lista de espaços ocupados na memória com dois objetos contidos dentro de si, representando os espaços tomados na memória pelas alocações anteriores. E a Figura 31 apresenta a lista de espaços disponíveis com dois trechos livres de memória não-contíguos.

Figura 31 - Memória com lista livre separada em dois trechos não-contíguos



Fonte: Próprio autor

Se os seus endereços fossem, de fato, contíguos então eles deveriam ter sido fundidos em um único objeto **MemoryChunk**. A fusão teria ocorrido se o objeto *struct*

tivesse saído da memória ao invés do inteiro alocado inicialmente. E acontecerá em seguida se, a partir do estado da Figura 31, a *struct* alocada seja “desalocada”. Neste caso a memória volta para o estado inicial ilustrado na Figura 29.

4.3.3 Funcionamento do Gerenciador de Memória

A forma como o simulador funciona foi pensada a partir do problema de como interpretar as funções de alocação e liberação de memória básicas do C, conforme apresentadas na Seção 2.3.4. Foram implementadas apenas as funções *malloc* e *free*, deixando as mais elaboradas para trabalhos futuros.

A memória só é alocada de fato quando o retorno da função *malloc* é um ponteiro que não é nulo para uma posição de memória. Portanto é importante salientar que as listas contidas no objeto **Heap** não são manipuladas durante a análise sintática, mas sim durante a execução do código intermediário. A não ser que se considere a chamada do método que reverte a memória ao seu estado inicial como uma manipulação.

Durante a execução do *front end* as funções são reconhecidas como funções de bibliotecas do C, como já ocorre atualmente com a *printf* e a *scanf*. E de fato o são, pertencendo à biblioteca *stdlib*. A análise sintática gera um comando de atribuição expressando o retorno da mesma através de um objeto do tipo **NodoExp**. Esse, por sua vez, conforme necessidade levantada apenas durante a etapa de implementação, delega as funções de alocação e liberação de memória novamente para a classe **Intermediario**.

Ao executar a linha de código intermediário a expressão é avaliada de acordo com a função de *cast* realizada sobre o retorno da função, utilizada para possibilitar o uso do ponteiro de memória como um tipo primitivo ou complexo através da variável alvo da atribuição. Dessa forma, é avaliado o que de fato se deve retornar e quais tipos de instâncias de **OTipo** devem ser criadas. Esta por sua vez, será atribuída como referência ao ponteiro que está do lado esquerdo da atribuição.

O equivalente da função *free* apenas busca o trecho de memória associado à variável e realiza o processo de liberação da memória através dos métodos da classe **Heap** definidos na Seção 4.3.2.

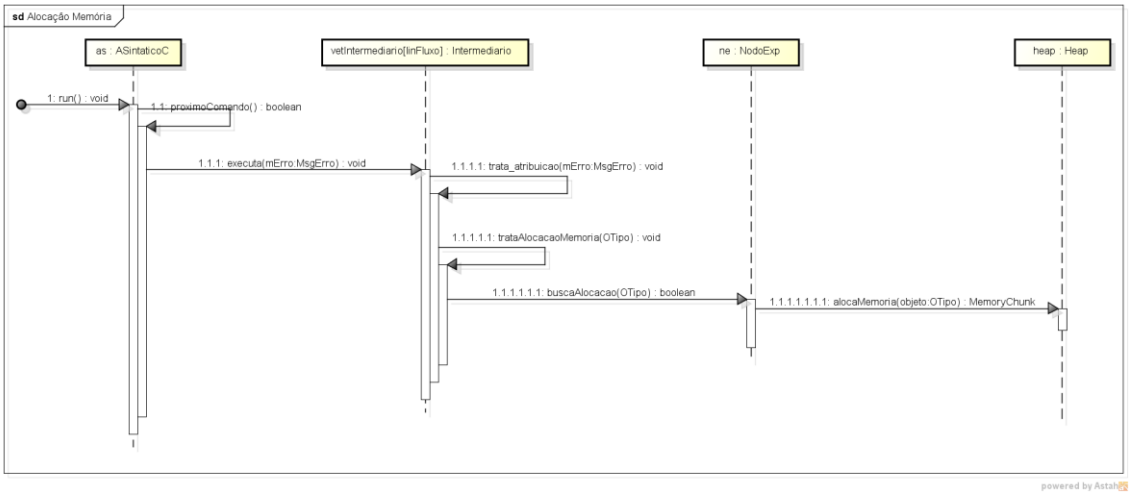
Considerando as duas funcionalidades necessárias, construídas a partir da chamada de diversos métodos em classes diferentes, conclui-se que a busca que eles

devem realizar em memória tem naturezas bem distintas, não podendo ser expressadas em um único diagrama ou bloco de código.

Enquanto a alocação consiste em buscar na lista de espaços disponíveis um espaço grande o bastante para acomodar o espaço de memória requisitado, o de “desalocação”, por sua vez, realiza uma busca em outra lista já tendo recebido o trecho de memória com o qual vai trabalhar. Após encontrar o local onde o espaço que está sendo desocupado deve ser colocado ainda deve ser realizado o processo de fusão (ou *merge*, para usar o termo em inglês) de dois ou até três trechos de memória diferentes, incluindo o que está sendo liberado, se necessário.

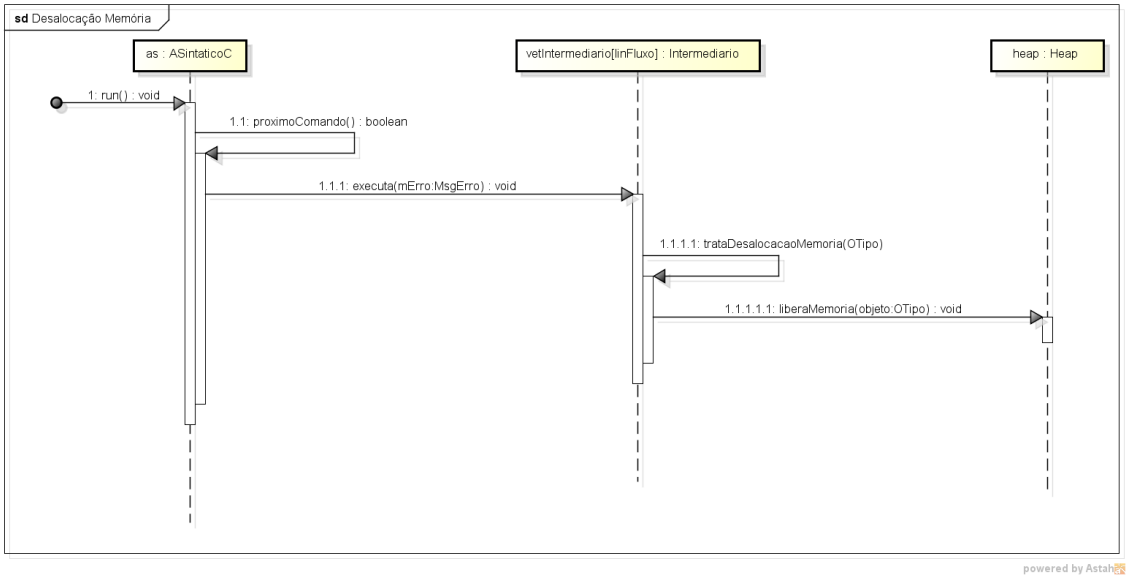
Os processos descritos estão ilustrados como diagramas de sequência nas Figuras 32 e 33. Seu objetivo é demonstrar a forma como as classes colaboram entre si, através de chamadas de métodos, para a execução dos processos de alocação e “desalocação” (Medeiros, 2004). Ambos foram atualizados após o desenvolvimento, conforme descrito na Seção 6.4.

Figura 32 - Diagrama de Sequência para alocação de memória



Fonte:Próprio autor

Figura 33 - Diagrama de Sequência para liberação de memória



Fonte: Próprio autor

5 METODOLOGIA DE DESENVOLVIMENTO

5.1 Incrementos e correções orientados a testes

Além da base teórica construída na primeira etapa do projeto e da base bibliográfica apresentada então, também foi importante o desenvolvimento das funcionalidades através da comparação com o funcionamento de outros compiladores e interpretadores. Neste caso foram utilizados tanto compiladores tradicionais como alguns interpretadores disponíveis online, todos os quais foram citados na Seção 2.3.1.

Cada nova funcionalidade no WebAlgo, antes mesmo do início da sua codificação, é expressa em uma bateria de testes construídos e executados em algum dos compiladores ou interpretadores já disponíveis para uso.

São realizadas comparações observando o comportamento que um determinado algoritmo executando um caso de testes apresenta em um compilador/interpretador já existente e no WebAlgo, onde as alterações necessárias para reconhecimento das novas construções sintáticas acabaram de ser implementadas. Portanto, a funcionalidade pode ser considerada como implementada se a bateria de testes for compilada e executada com sucesso pelo WebAlgo.

A avaliação comparativa da execução se dá através de uma observação comportamental a nível de usuário, pois a lógica interna das ferramentas já existentes é desconhecida em todos os níveis. Mais formalmente, tratam-se de testes de caixa preta.

Adicionalmente são construídos alguns casos que devem gerar erros de compilação ou de execução, ou seja, uma falha expressamente prevista em código no analisador sintático ou no interpretador de código intermediário. Devido ao caráter didático do WebAlgo é importante que, de fato, todos os possíveis erros dessas duas naturezas distintas estejam expressamente previstos no código de forma a fornecer ao usuário mensagens informativas sobre o problema encontrado.

Observe-se que a diferença entre erros de compilação ou de execução não é correspondente à diferença entre erros de sintaxe e de semântica. Se um erro de semântica pode ser identificado durante a fase de análise sintática é a política deste projeto e a opinião aqui defendida de que ele deve ser apontado para o usuário

naquele momento ao invés de passar à etapa de execução com a certeza de que algum comportamento inesperado pode ocorrer.

A implementação e manutenção dessas baterias de testes é uma técnica que já vinha sendo utilizada no desenvolvimento do WebAlgo antes deste projeto. É especialmente útil para testes de regressão.

Houve um problema que foi identificado através de testes de regressão realizados com o uso da interface gráfica do WebAlgo. Observe-se que a maior parte deste projeto foi desenvolvida utilizado um testador automático sem interface. Em essência, não há diferença na utilização de nenhum dos dois. Maiores informações a respeito, com figuras ilustrativas, estão disponíveis na Seção 6.1.

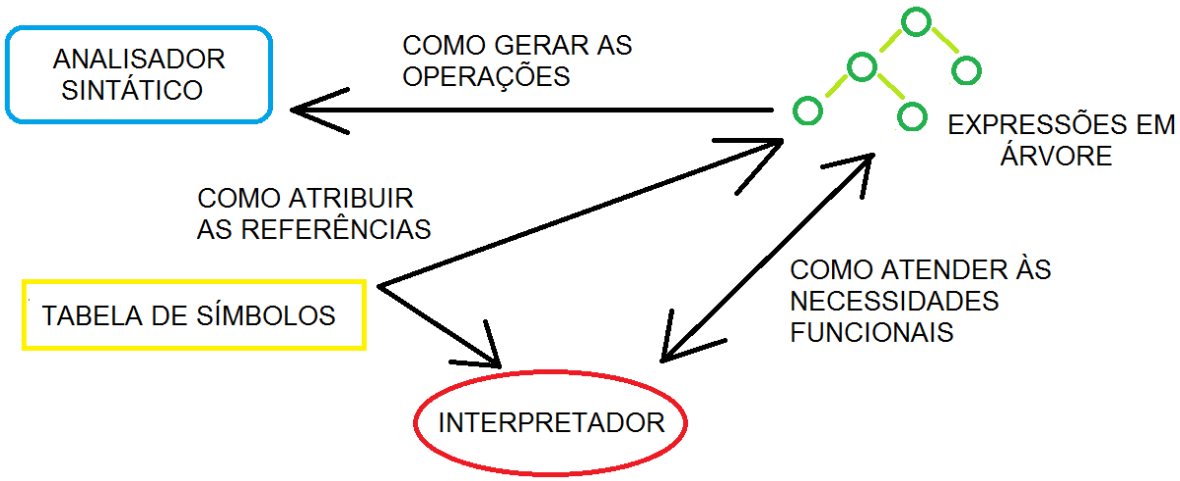
O caso em questão identificou que uma alteração realizada no motor de execução para permitir a coexistência da interface gráfica e do testador automático local desenvolvido para este projeto gerou um erro inesperado durante a execução de alguns casos de testes básicos. O problema foi facilmente identificado através do uso da ferramenta de depuração do Eclipse, amplamente utilizada durante o desenvolvimento. Esse caso ilustra a importância da possibilidade de testes automatizados em projetos de compiladores/interpretadores.

5.2 Fluxo de alterações incrementais

A implementação incremental das alterações guiadas pelo uso dos casos de testes funcionou bem com o fluxo de alterações incremental utilizado no projeto. Este é, e sempre deve ser baseado em um entendimento do funcionamento de um compilador/interpretador. Não apenas conceitualmente como também de acordo com as particularidades do caso em questão. Neste caso, o fluxo aqui está analisado e descrito com base no caso particular do WebAlgo.

Em síntese o que ocorre é que o analisador sintático além de realizar a verificação da sintaxe também deve gerar as estruturas intermediárias. A maior parte das alterações realizadas no interpretador ocorreram na classe **NodoExp**, cuja responsabilidade é formar e resolver a árvore de operadores e operandos das expressões reconhecidas. Portanto o trabalho de codificação consistiu em tratar no analisador sintático todas as construções sintáticas que passaram a ser reconhecidas nesta versão e gerar a árvore de expressões de forma a traduzir corretamente a semântica para posterior interpretação. A Figura 34 ilustra o fluxo das alterações.

Figura 34 – Fluxo ilustrativo de uma alteração incremental



Fonte: Próprio autor

É desejável que a primeira alteração ocorresse na *engine* de resolução de expressões. Dessa forma já é possível definir como a árvore deve estar montada para garantir a interpretação correta. O método *buscaValor* da classe **NodoExp** concentrou a maior parte dessas alterações.

Depois vêm o desenvolvimento dos incrementos e adaptações na classe **ASintaticoC**. Quando adaptações foram necessárias frequentemente a identificação dos locais que deveriam ser alterados dentro da classe só foi possível após algumas rodadas de testes.

Quanto ao desenvolvimento das estruturas de apoio, este poderia ter ocorrido antes ou depois do processo descrito. Estas, juntamente com outras já existentes no WebAlgo e que tiveram de ser adaptadas em diversos sentidos, tais como as classes **OTipo, ConteudoVariaveis** e **OFuncao**, formam uma espécie de tabela de símbolos dispersa. Observe-se que esse aspecto disperso não é necessariamente ruim, inclusive permitindo uma maior versatilidade e usabilidade.

Na prática o que ocorreu foi que toda a estrutura planejada anteriormente foi desenvolvida antes, incluindo os mapas de *structs* e a versão original do mapa de tipos simples. Com a continuidade do desenvolvimento e dos testes novas necessidades surgiram e, portanto, adaptações tiveram de ser feitas, incluindo algumas novas classes. Nominalmente, a **IdPosALexico**, **PosicaoALexico** e a **TipoSimples**.

6 FUNCIONALIDADES IMPLEMENTADAS

6.1 TESTADOR AUTOMÁTICO

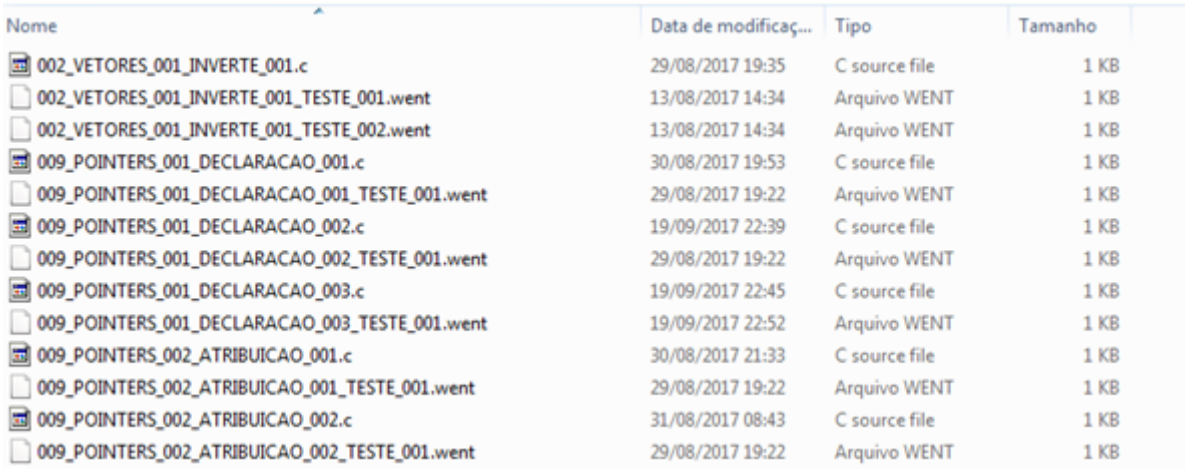
Foi desenvolvida uma ferramenta que realiza a análise de todos os programas em C existentes em um determinado diretório e depois os executa respeitando os dados fornecidos por casos de testes também disponíveis dentro do mesmo diretório. Cada um dos algoritmos deve ser um programa em C devidamente escrito dentro dos padrões sintáticos e semânticos, tendo a extensão “.c”.

Por padrão, todos os casos de testes disponíveis para um determinado algoritmo devem seguir um padrão de nomenclatura que permita identificar a qual programa ele pertence. E todos eles devem ter a extensão “.went”.

Se o nome do programa em questão é “programa.c” então os seus casos de testes devem estar nomeados tendo o nome do arquivo “.c” correspondente como sufixo. Neste caso os casos de testes poderiam estar nomeados como “programa_1.went”, “programa_2.went” e “programa_3.went”, por exemplo.

O diretório apresentado na Figura 35 ilustra o formalismo dos nomes dos arquivos onde os casos de testes referentes a um programa estão logo abaixo do mesmo devido à ordenação aplicada sobre os arquivos. Esses padrões simples facilitaram o gerenciamento dos testes.

Figura 35 - Lista de algoritmos e seus respectivos casos de teste



Nome	Data de modificaç...	Tipo	Tamanho
002_VETORES_001_INVERTE_001.c	29/08/2017 19:35	C source file	1 KB
002_VETORES_001_INVERTE_001_TESTE_001.went	13/08/2017 14:34	Arquivo WENT	1 KB
002_VETORES_001_INVERTE_001_TESTE_002.went	13/08/2017 14:34	Arquivo WENT	1 KB
009_POINTERS_001_DECLARACAO_001.c	30/08/2017 19:53	C source file	1 KB
009_POINTERS_001_DECLARACAO_001_TESTE_001.went	29/08/2017 19:22	Arquivo WENT	1 KB
009_POINTERS_001_DECLARACAO_002.c	19/09/2017 22:39	C source file	1 KB
009_POINTERS_001_DECLARACAO_002_TESTE_001.went	29/08/2017 19:22	Arquivo WENT	1 KB
009_POINTERS_001_DECLARACAO_003.c	19/09/2017 22:45	C source file	1 KB
009_POINTERS_001_DECLARACAO_003_TESTE_001.went	19/09/2017 22:52	Arquivo WENT	1 KB
009_POINTERS_002_ATRIBUICAO_001.c	30/08/2017 21:33	C source file	1 KB
009_POINTERS_002_ATRIBUICAO_001_TESTE_001.went	29/08/2017 19:22	Arquivo WENT	1 KB
009_POINTERS_002_ATRIBUICAO_002.c	31/08/2017 08:43	C source file	1 KB
009_POINTERS_002_ATRIBUICAO_002_TESTE_001.went	29/08/2017 19:22	Arquivo WENT	1 KB

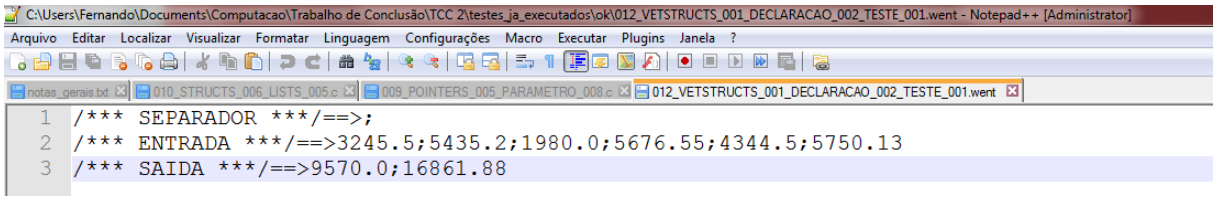
Fonte: Próprio autor

O testador automático está codificado na classe **AutomaticTester**, que também possui um método *main* que permite ao programador que esteja alterando o WebAlgo rodar testes de compilação e interpretação sem precisar acessar a interface gráfica. Foi criado um novo método chamado *rodaSincrono* na classe **ASintaticoC** que permite executar o algoritmo várias vezes seguidas.

Cada uma dessas execuções roda um dos vários casos de testes possivelmente definidos para o algoritmo no diretório. Se não houver nenhum então o programa simplesmente não é executado e o testador passa a iniciar o reconhecimento do próximo algoritmo.

Em um dos casos de testes definidos é obrigatório informar a entrada e a saída de dados esperadas. Ambos são obrigatórios. E também é possível informar outro separador para possibilitar ao WebAlgo identificar onde termina uma entrada e saída e onde começa outra. Se não informado o *default* é a vírgula simples. A Figura 36 apresenta um exemplo de um caso de testes.

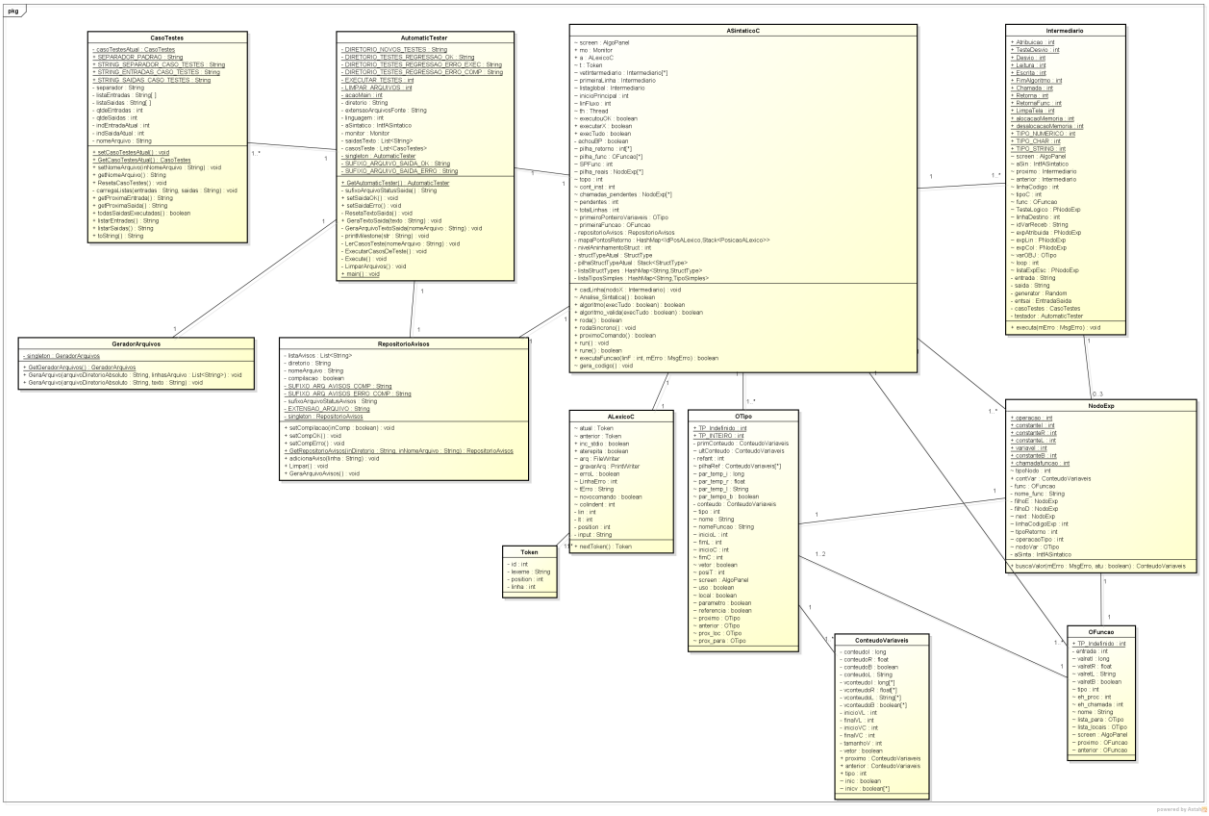
Figura 36 - Exemplo de um caso de testes usado no Testador Automático



Fonte: Próprio autor

A Figura 37 apresenta um diagrama de classes simplificado que mostra como o testador se relaciona com as principais classes do compilador/interpretador.

Figura 37 - Diagrama de Classes simplificado do Testador Automático

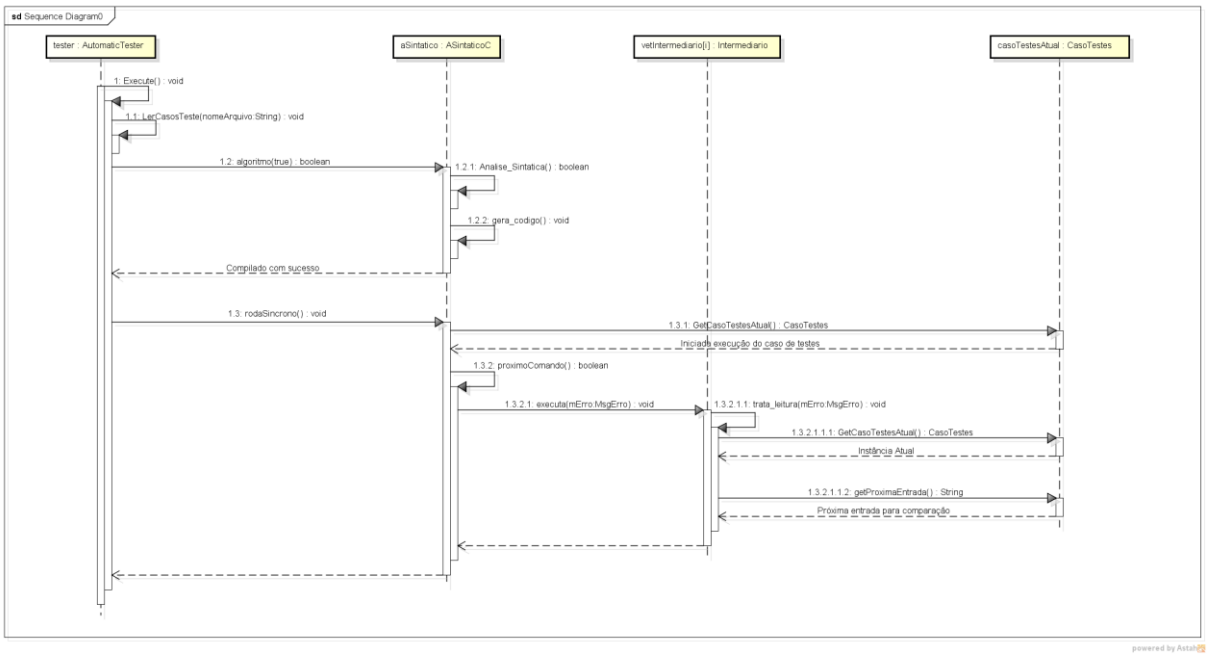


Fonte: Próprio autor

Após a análise sintática ocorre a leitura de todos os casos de testes disponíveis para o programa recém reconhecido através do método *LerCasosTeste*, da classe **AutomaticTester**. Estes são armazenados em uma lista de instâncias da classe **CasoTestes** e se alternam para ocupar o atributo estático *casoTestesAtual* da classe, pois este indica qual caso está sendo executado no momento. Esta foi parcialmente inspirada na já existente anteriormente **EntradaSaida**, mas com algumas adaptações. Uma comparação rápida entre as duas classes confirmará essa afirmação. Os casos de testes automatizados devem substituir as entradas e saídas informadas através da execução do programa.

O diagrama da Figura 38 demonstra como ocorre a troca de mensagens entre os objetos para a execução dos testes automáticos e validação dos casos de testes.

Figura 38 - Diagrama de Sequência simplificado do Testador Automático



Fonte: Próprio autor

Portanto, o caso de testes atual é utilizado nos métodos de leitura e escrita da classe **Intermediario** para ocupar o lugar dos testes fornecidos por uma instância da **EntradaSaida** nos casos em que o testador automático não está sendo utilizado. Estes são identificados pela inexistência da referência da classe **AlgoPanel**, que é a instância da interface gráfica do WebAlgo.

Os construtores do analisador sintático exigem, por parâmetro, a instância da interface. Para que fosse possível realizar todo o processo sem o uso da mesma

vários métodos tiveram de ser alterados em praticamente todas as classes do programa. Isso porque a sua referência é repassada para as linhas de código intermediário, possibilitando a sua interação com a mesma para a execução dos comandos de leitura e escrita.

Uma sugestão futura para alteração seria não mais passar essa instância como parâmetro e aplicar à classe **AlgoPanel** o padrão de projetos *Singleton*, de mesma forma que foi implementado neste projeto nas novas classes **AutomaticTester**, **Heap**, e várias outras. Este padrão é muito útil neste caso pois permite uma instância única da classe, que é o caso em questão, mas com um ponto global de acesso. O objeto pode ser buscado de qualquer local sempre que necessário.

6.2 IMPLEMENTAÇÃO DE PONTEIROS

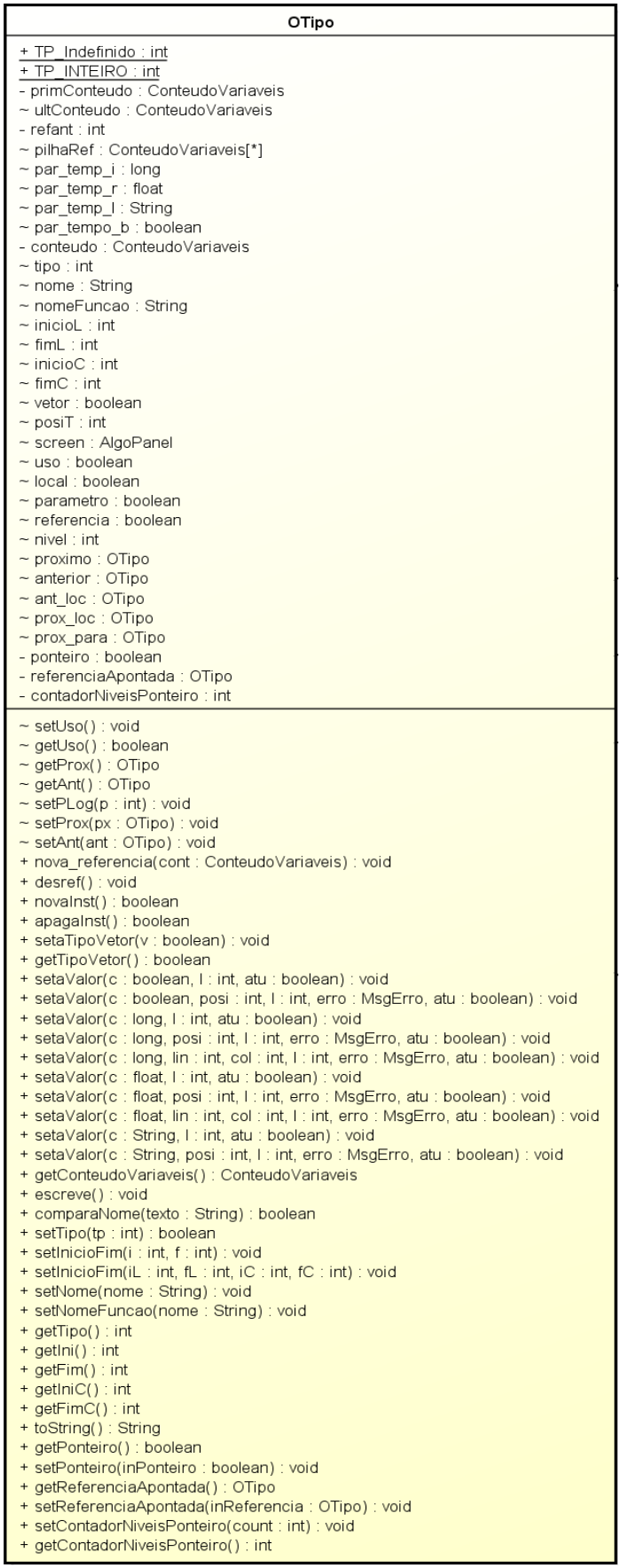
Como já mencionado anteriormente a estrutura intermediária representativa utilizada para os ponteiros foi a definida na etapa de projeto, conforme descrito anteriormente na Seção 5.1. A principal dificuldade com relação ao funcionamento da mecânica dos ponteiros pode ser ilustrada pelo funcionamento do método *buscaValor* da classe **NodoExp**.

O WebAlgo estava preparado para que uma variável recebesse um valor, mas não uma referência para outra variável, ou seja, uma instância da classe **OTipo**. O método em questão retornava, e ainda retorna, uma instância da classe **ConteudoVariaveis**. Havia, portanto, a importante pendência de como tratar a funcionalidade do operador de endereço ‘&’.

A solução encontrada foi fazer com que a instância de **ConteudoVariaveis** retornada pelo método *buscaValor* pudesse armazenar a referência para uma variável. Assim, foi criado o atributo *referenciaApontada* para a classe e os respectivos métodos de busca e alteração do mesmo. Uma instância da classe que carrega uma referência para uma variável pode ser identificada pelo tipo *Constantes.Tp_Op_Endereco*.

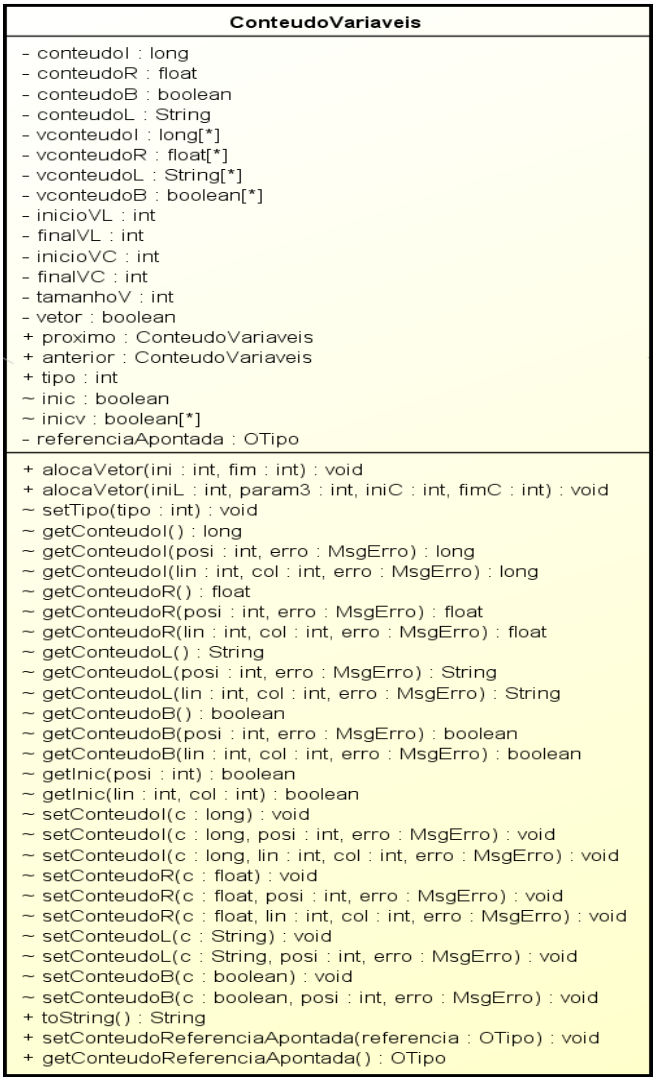
A Figura 39 mostra a nova estrutura da classe **OTipo** com as alterações necessárias para implementação de ponteiros. E a Figura 40 mostra a nova estrutura da classe **ConteudoVariaveis** após ser alterada para a mesma finalidade.

Figura 39 - Adaptações em OTipo para implementação de ponteiros



Fonte: Próprio autor

Figura 40 - Adaptações em ConteudoVariaveis para implementação de ponteiros



Fonte: Próprio autor

Toda a mecânica de atribuição de valores para as variáveis ocorre dentro da classe **NodoExp**, com exceção das novas tratativas de alocação de memória que são tratadas de forma especial na classe **Intermediario**. A mecânica de atribuição de endereços teve de ser desenvolvida em duas frentes, cada uma das quais representando um dos lados de uma atribuição. Portanto, as alterações necessárias foram basicamente realizadas nos métodos especializados *trata_var* e *trata_op*.

Um tratamento similar ocorre para o operador de “indireção” ‘*’, com a diferença de que este pode ser aplicado inúmeras vezes sobre uma variável, e por isso o nodo construído para a variável em questão durante a análise sintática guarda um contador de utilização do operador. Na execução do código intermediário quando for necessário buscar o valor dessa variável a operação de “indireção” deve ser resolvida.

Isso ocorre de forma recursiva, onde para cada uso do operador de “indireção” ocorre a busca da referência para a qual a variável resolvida anteriormente estava apontando. Se houver mais um operador depois deste, ele vai buscar a variável para a qual o ponteiro obtido na resolução imediatamente anterior estava apontando.

Por exemplo, se o operador de “indireção” é aplicado duas vezes sobre uma variável *a*, que deve necessariamente ser um ponteiro para um ponteiro, então o interpretador, dentro do método *trata_var* da classe **NodoExp**, deve buscar o valor da variável apontada pelo ponteiro, que por sua vez é apontado por *a*. Supondo que *a* aponte para *b*, que deve ser um ponteiro simples, então o valor considerado é o valor atual da variável para a qual *b* está apontando.

Algoritmo 6 – Exemplo do uso de múltiplos operadores de indireção

```c
#include <stdio.h>

int main(){

	int i, *j, **p;
	scanf("%d",&i);

	j = &i;
	p = &j;
	printf("%d",**p);

	return 0;
}
```

Fonte: Próprio autor

Ao executar o Algoritmo 6 no WebAlgo o valor de saída será o mesmo da entrada devido à implementação da funcionalidade de resolução de indireção no interpretador. A expressão ‘***p*’ é representada por um único nodo na árvore de expressões - uma instância de **NodoExp** - configurado com dois níveis de indireção, conforme determina a sintaxe. Um dos níveis é resolvido encontrando a variável *j* a partir de *p* e o outro encontrado *i* a partir de *j*.

Durante essa operação a variável apontada pelo nodo é alterada. Para isso foi criada a variável *nodoVarResolvida* usada no método *trata_var* da classe **NodoExp**. Também serve de forma análoga para os casos em que ocorre o uso do operador ‘->’ para buscar membros de *structs* apontadas por ponteiros.

Essa nova variável é definida dentro do referido método através de uma nova função especializada denominada *resolveVariavel* chamada no início do mesmo. Esta é responsável por resolver o(s) operador(es) de “indireção” e as operações realizadas sobre ponteiros para *structs* nos casos em que não é possível resolver a referência desejada pelo usuário durante a análise sintática, ou “em tempo de compilação”, coloquialmente, já que não se sabe qual instância de **OTipo** será apontada por um ponteiro. É essencialmente o mesmo problema apresentado pelos operadores de “indireção”. Mais sobre a resolução de campos de *structs* na Seção 6.3.

Durante o processo de análise sintática os diversos operadores de “indireção” utilizados sobre uma variável, ou em quaisquer tipos de declarações são colhidos através de um laço simples e repassados para as estruturas intermediárias através de métodos *set*. Isso define para o interpretador qual o “nível de indireção” que deve ser aplicado sobre uma variável. É então resolvido de forma recursiva, conforme explicado anteriormente.

Além disso também é possível comparar ou atribuir um ponteiro em C com uma referência nula. A atribuição nula para um ponteiro faz com que este aponte para lugar nenhum, ou seja para nenhum endereço de memória. Comparar o ponteiro com uma referência nula verifica se ele está apontando para algum endereço no momento da comparação.

Isso é possível com o uso da palavra reservada *NULL*. Toda a lógica necessária para simular o funcionamento de comparações usando o *NULL* que resultam em valores lógicos e atribuição do *NULL* para ponteiros ou outros tipos de variáveis dentro do interpretador estão tratadas dentro da classe **NodoExp**, especificamente nos métodos *trataOperacoesComparativas* e *trata_var*, respectivamente.

Alguns compiladores consideram ponteiros não inicializados como *NULL* e outros não, dependendo de configurações. Mas o correto é que todas as variáveis devem ser inicializadas com zero ou equivalente. Para ponteiros, portanto, será o valor NULL, tipicamente definido como *0* ou *((void *) 0)*, de acordo com (Summit).

Portanto, no WebAlgo, ponteiros não inicializados serão considerados nulos, ou seja, considerados como tendo o valor *null pointer*.

6.3 IMPLEMENTAÇÃO DE STRUCTS

Dentre as três grandes funcionalidades descritas nesta Seção 6, com exceção do Testador Automático, que é acessório para os propósitos do projeto, a implementação de *structs* foi a que necessitou muito mais desenvolvimento do que o previsto originalmente.

Estruturas de controle foram criadas na classe **ASintaticoC** na forma de mapas de instâncias da classe **StructType** para guardar *structs* e tipos definidos com a palavra reservada *typedef*. Uma struct que se torna um tipo complexo a partir do *typedef* é guardada no mesmo mapa das *structs* declaradas sem o uso da palavra reservada e que não foram “tipadas” posteriormente. Entretanto, há uma diferença no método de inserção e busca das declarações para satisfazer os usos da sintaxe com e sem o uso de tipos definidos pelo usuário. O Algoritmo 7 apresenta os métodos de busca de *structs* utilizados durante a análise sintática.

Foi desenvolvida uma funcionalidade semelhante na classe **OFuncao** para que se tenha uma separação de escopos implementada de uma forma simples, como é possível perceber através da análise dos métodos. Esses métodos buscam a estrutura complexa declarada recebendo como parâmetro o nome definido para o tipo ou o nome da *struct*.

A *struct* em questão pode ter apenas um nome, que é o identificador que vem logo após a palavra reservada *struct* na declaração, mas pode estar “tipada” com diversos nomes. O que ocorre então é que a mesma estrutura poderá estar presente diversas vezes no mesmo mapa de *structs* com diferentes chaves de mapeamento.

No entanto todas as entradas correspondentes à mesma estrutura dentro do mapa, não importando qual a chave, apontando para a mesma instância de **StructType** garantindo que todas as declarações realizadas, não importando qual o nome do tipo utilizado, serão referentes à mesma *struct*. O Algoritmo 8 é um exemplo de um programa em que a mesma *struct* terá múltiplas entradas dentro do mapa de tipos complexos se ignorados os diferentes escopos.

Nesse caso as instâncias s1 e s2 são globais enquanto s3 e s4, respectivamente dos tipos tp_st1 e tp_st2, que são duas “tipagens” da mesma *struct*, estão restritos ao escopo da função *main*. Se o programa necessitasse de mais uma instância da *struct* st1 haveriam três formas de declará-la: através do nome da *struct* ou de um dos dois *types* definidos com o uso do *typedef*.

Algoritmo 7 – Métodos de busca de structs da classe ASintaticoC

```
private HashMap<String,StructType> listaStructTypes;

private StructType buscaStructPorNome(String nomeStruct){
	if (this.listaStructTypes == null) return null;
	return this.listaStructTypes.get("struct " + nomeStruct);
}

private StructType buscaStructPorNome(String nomeStruct, OFuncao funcao){
	if (funcao == null) return this.buscaStructPorNome(nomeStruct);
	else {
		StructType st = funcao.buscaStructPorNome(nomeStruct);
		if (st == null)
			st = this.buscaStructPorNome(nomeStruct);
		return st;
	}
}

private StructType buscaStructPorTipo(String nomeTipoComplexo){
	if (this.listaStructTypes == null) return null;
	return this.listaStructTypes.get(nomeTipoComplexo);
}

private StructType buscaStructPorTipo(String nomeTipoComplexo, OFuncao
funcao){
	if (funcao == null)
		return this.buscaStructPorTipo(nomeTipoComplexo);
	else {
		StructType st = funcao.buscaStructPorTipo(nomeTipoComplexo);
		if (st == null)
			st = this.buscaStructPorTipo(nomeTipoComplexo);
		return st;
	}
}
```

Fonte: Próprio autor

Algoritmo 8 – Programa com múltiplos mapeamentos para a mesma struct

```c
#include <stdio.h>
#include <string.h>

struct st1 {
	int id;
	char nome [10];
	float vlr;
} s1, s2;

typedef struct st1 tp_st1, tp_st2;

int main(){
	scanf("%d",&s1.id);
	scanf("%s",s1.nome);
	scanf("%f",&s1.vlr);

	s2.id = s1.id/2;
	strcpy(s2.nome, s1.nome);
	s2.vlr = s1.vlr * 0.75;

	printf("%d",s2.id);
	printf("%s",s2.nome);
	printf("%f",s2.vlr);

	tp_st1 s3;
	s3.id = s2.id * 3;
	strcpy(s3.nome, "TestSt");
	s3.vlr = strlen(s2.nome);

	printf("%d",s3.id);
	printf("%s",s3.nome);
	printf("%f",s3.vlr);

	tp_st2 s4;
	s4.id = s2.id * 3;
	strcpy(s4.nome, "TestSt");
	s4.vlr = strlen(s2.nome);

	printf("%d",s4.id);
	printf("%s",s4.nome);
	printf("%f",s4.vlr);

	return 0;
}
```

Fonte: Próprio autor

Quanto ao uso do *typedef* para definir tipos que não são *structs*, por conveniência nomeados no projeto como tipos não complexos, ou tipos simples, foi criada uma classe denominada **TipoSimples**. Esta é utilizada nas classes **ASintaticoC** e **OFuncao** exatamente da mesma forma que a **StructType** é utilizada para *structs*, conforme explicado anteriormente.

A primeira implementação utilizada consistia em um mapa de valores inteiros que identificavam o tipo primitivo atribuído ao tipo não complexo através do identificador da classe **Constantes**. Uma revisão concluiu que o tipo pode ser um ponteiro, vetor ou matriz, portanto a identificação da variável envolve mais do que o seu tipo primitivo. Conclui-se que existe a necessidade um objeto com diversos atributos para possibilitar a tratativa de todos os diferentes casos suportados pela linguagem. Daí a necessidade da implementação da classe **TipoSimples**.

Outra necessidade, que é parcialmente relacionada com a possibilidade de definir diversos nomes para um mesmo tipo complexo é de que o interpretador deve ser capaz de atribuir uma *struct* para outra diretamente, contanto que ambas as instâncias sejam da mesma *struct*.

Modificando parcialmente o Algoritmo 8 podemos obter o Algoritmo 9 para ilustrar as atribuições válidas na linguagem C. Uma execução dos exemplos dentro do WebAlgo, com a ajuda do Testador Automático, se necessário, comprova que ele está de acordo com as definições corretas.

Os casos implementados no Algoritmo 9 são de instâncias e tipos que fazem referência à mesma declaração de *struct*. A atribuição direta não é válida, no entanto, para qualquer caso em que os tipos envolvidos não sejam derivados da mesma estrutura, isto é, todas as instâncias envolvidas devem ser da mesma *struct*.

Isso significa que mesmo que existam duas *structs* distintas, declaradas separadamente, que sejam idênticas, isto é, com o mesmo número de campos, todos do mesmo tipo e com os mesmos nomes a atribuição direta não é válida porque as duas instâncias não são do mesmo tipo. A linguagem C não realiza uma comparação interna dos membros da *struct* um a um. Ele apenas define os tipos customizados e de cada uma das instâncias declaradas durante a declaração.

Algoritmo 9 – Programa com atribuições diretas de structs

```c
#include <stdio.h>
#include <string.h>

struct st1 {
	int id;
	char nome [10];
	float vlr;
} s1, s2;

typedef struct st1 tp_st1, tp_st2;

int main(){
	scanf("%d",&s1.id);
	scanf("%s",s1.nome);
	scanf("%f",&s1.vlr);

	s2.id = s1.id/2;
	strcpy(s2.nome, s1.nome);
	s2.vlr = s1.vlr * 0.75;

	printf("%d",s2.id);
	printf("%s",s2.nome);
	printf("%f",s2.vlr);

	tp_st1 s3;
	s3.id = s2.id * 3;
	strcpy(s3.nome, "TestSt");
	s3.vlr = strlen(s2.nome);

	printf("%d",s3.id);
	printf("%s",s3.nome);
	printf("%f",s3.vlr);

	tp_st2 s4;
	s4.id = s2.id * 3;
	strcpy(s4.nome, "TestSt");
	s4.vlr = strlen(s2.nome);

	printf("%d",s4.id);
	printf("%s",s4.nome);
	printf("%f",s4.vlr);

	return 0;
}
```

Fonte: Próprio autor

Toda e qualquer instância de uma *struct* dentro do WebAlgo é criada pelo método *retornaInstancia* da classe **StructType**. Portanto quando a variável da *struct* é criada já está garantido que cada um dos seus membros tenha o seu próprio objeto guardado no mapa de campos membros.

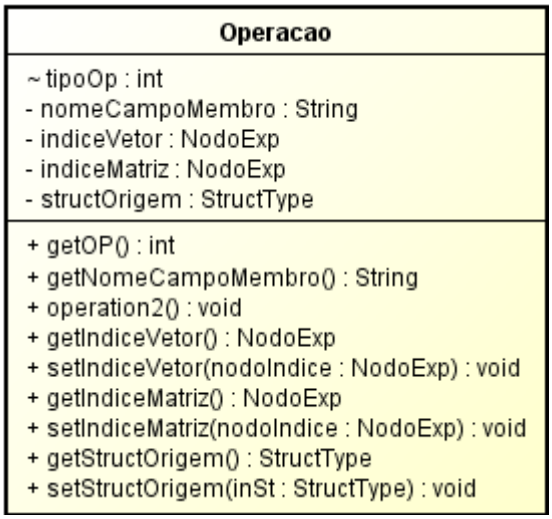
Após a declaração, sempre que um campo da *struct* é referenciado através do operador ‘.’ caberá ao analisador sintático resolver a referência verificando se o identificador que vier após o ponto identifica um dos campos membros da *struct*. Em caso positivo a nova instância de **OTipo** retornada pelo método *retornaCampo*, chamado a partir da instância original que identifica a *struct*, será a variável informada para as estruturas intermediárias.

Mas, no caso em que o operador ‘->’, em se tratando, portanto, de ponteiros para *structs* não é possível que o analisador sintático tenha ciência de qual será a variável apontada pelo ponteiro já que as estruturas intermediárias não podem refletir uma realidade que só vai ocorrer em tempo de execução. Foi necessário, para sanar esse problema que se criasse um novo tipo de estrutura atrelada ao nodo instância de **NodoExp** que representa a variável em questão.

Na classe **ASintaticoC** foi criado o método *resolveMembroStruct* responsável por retornar qual a variável referenciada, quando possível. Este também é utilizado em casos em que vetores de *structs* são utilizados e para o reconhecimento do operador ‘.’, reunindo a lógica para reconhecer qualquer referência dentro de uma instância de *struct*, não importando a dimensão definida ou o nível de encadeamento ao qual pertence um campo referenciado.

O *resolveMembroStruct* é utilizado de forma recursiva para reconhecer, sucessivamente, um membro de um elemento dentro de um vetor de *structs*, ou um elemento de um membro, ou ainda membro de um membro, etc. Utiliza para tal um atributo da classe, através do método *enfileirarOperacaoStruct* para guardar todas as operações solicitadas que não podem ser resolvidas pelo analisador, na ordem em que foram definidas no código fonte. Posteriormente toda a fila é transferida ao nodo que representa a variável na árvore da expressão construída com instâncias da classe **NodoExp**. Esta fila é formada de objetos da classe **Operacao** que foi expandida para acomodar as novas funcionalidades necessárias. A Figura 41 mostra a nova estrutura da classe.

Figura 41 - Estrutura da classe Operacao



Fonte: Próprio autor

Durante a interpretação do programa as operações coletadas são resolvidas pelo método *resolveOperacoesStruct* invocado dentro do *resolveVariavel*, que já foi citado na Seção 6.2.

O apêndice D apresenta o método na íntegra, onde é possível verificar o uso das estruturas intermediárias durante o processo de interpretação e a lógica desenvolvida para localizar, através das referências encadeadas, a instância de **OTipo** que representa a variável requisitada no programa fonte.

Ele resolve cada operação, uma por uma, redefinindo a *nodoVarResolvida* em cada nova iteração. É conceitualmente similar ao que ocorre com a lógica usada para os operadores de “indireção” no método *resolveReferenciaIndirecao*, porém mais complexo devido aos diferentes tipos de operadores que deve suportar.

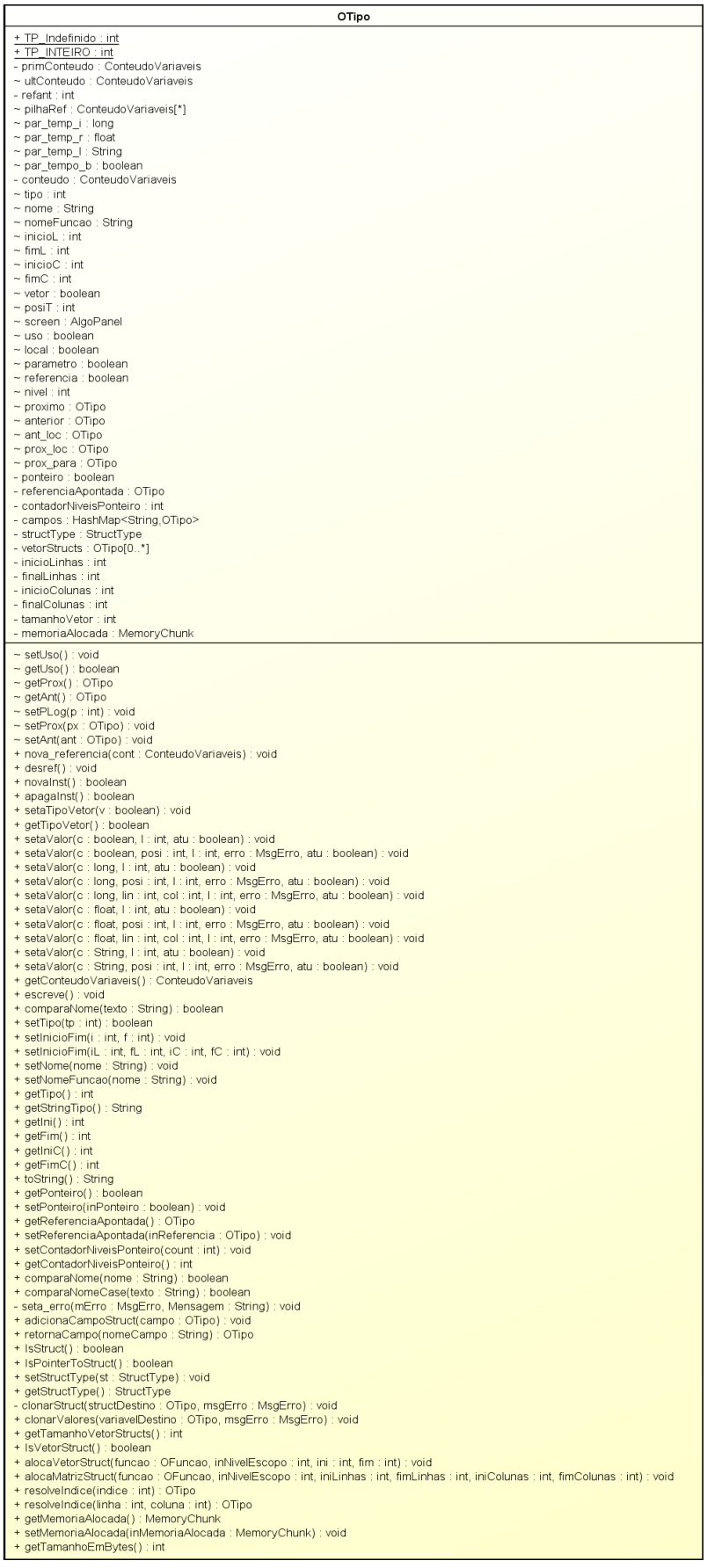
As referências a vetores de *structs* são reconhecidas no analisador sintático através do método *resolveMembroStruct* e interpretadas pelo método *resolveOperacoesStruct*, conforme destacado. Mas para isso foi necessária a criação de novos atributos e métodos dentro da classe **OTipo** que são similares ao que foi utilizado para vetores comuns na classe **ConteudoVariaveis**. Essa estrutura à parte foi necessária pois para os vetores de *structs* é necessário que cada elemento seja uma instância separada de **OTipo**. O Algoritmo 10 mostra as novas estruturas criadas e a Figura 42 apresenta a estrutura da classe **OTipo** contendo também todas as novas adaptações para implementação de *structs*.

Algoritmo 10 – Estrutura criada em OTipo para suporte a vetores de *structs*

```
private OTipo [] vetorStructs; private int tamanhoVetor;
private int inicioLinhas; private int finalLinhas;
private int inicioColunas; private int finalColunas;
public int getTamanhoVetorStructs(){
   if (this.tipo != Constantes.TK_Struct) return -1;
   if (vetorStructs == null) return -1;
   return this.tamanhoVetor;
}
public boolean IsVetorStruct(){
   if (this.tipo != Constantes.TK_Struct) return false;
   if (vetorStructs == null) return false;
   return this.vetorStructs.length > 0;
}

public void alocaVetorStruct (OFuncao funcao, int inNivelEscopo, int ini,
int fim){
   if (this.getTipo() != Constantes.TK_Struct) return;
   if (this.structType == null) return;
   this.campos.clear(); this.vetorStructs = new OTipo[tamanhoVetor];
   this.inicioLinhas = ini; this.finalLinhas = fim;
   this.inicioColunas = -1; this.finalColunas = -1;
   this.tamanhoVetor = fim-ini+1;
   for (int i=0; i<tamanhoVetor; i++)
      this.vetorStructs[i] = this.structType.retornaInstancia(funcao,
this.nome + "[" + i + "]", inNivelEscopo);
}

public void alocaMatrizStruct (OFuncao funcao, int inNivelEscopo, int
iniLinhas, int fimLinhas, int iniColunas, int fimColunas){
   if (this.getTipo() != Constantes.TK_Struct) return;
   if (this.structType == null) return;
   this.campos.clear(); this.vetorStructs = new OTipo[tamanhoVetor];
   this.inicioLinhas = iniLinhas; this.finalLinhas = fimLinhas;
   this.inicioColunas = iniColunas; this.finalColunas = fimColunas;
   this.tamanhoVetor = ( fimLinhas - iniLinhas + 1 ) * ( fimColunas -
iniColunas + 1 );
   for (int i=0; i < this.tamanhoVetor; i++)
      this.vetorStructs[i] = this.structType.retornaInstancia(funcao,
this.nome + "[" + i + "]", inNivelEscopo);
}
public OTipo resolveIndice (int indice) {
   if (indice >= tamanhoVetor) return null;
   return this.vetorStructs[indice];
}
public OTipo resolveIndice (int linha, int coluna){
   int pos= coluna - this.inicioColunas + ( linha - this.inicioLinhas ) *
( this.finalColunas - this.inicioColunas + 1 );
   if (pos >= tamanhoVetor) return null;
   return this.resolveIndice(pos);
}
```

Fonte: Próprio autor

Figura 42 - Estrutura final da classe OTipo



Fonte: Próprio autor

A implementação final que deve ser mencionada e é relacionada a *structs* corresponde ao que foi denominado no código como auto referências. Trata-se dos casos em que uma *struct* é declarada contendo um ou mais campos membros que são ponteiro para uma instância dessa mesma *struct*. Essa funcionalidade é fundamental para a construção de estruturas de dados tais como listas encadeadas e árvores. Portanto é altamente desejável que o WebAlgo suportasse o uso destes ponteiros.

O problema aqui era definir um objeto **StructType** para uma instância sem que ele já estivesse pronto e, além disso, essa mesma instância de **StructType** teria que conter uma referência a si mesma entre os seus membros. Isso causaria a criação de uma estrutura circular que, em uma recursão - e existem algumas dentro do projeto que iteram sobre os membros de uma *struct* - causaria um *loop* infinito.

A solução foi criar um clone do objeto **StructType** que representa a *struct* que está, no momento, sendo reconhecida e atribuir esse clone para o ponteiro declarado como membro. Para tal foi criado o método *clonar* e o *adicionaAutoReferencia*.

Ao finalizar o reconhecimento da *struct* é chamado o novo método *atualizaAutoReferencia* para assegurar que todos os clones cadastrados tenham todos os campos membros declarados.

6.4 GERENCIAMENTO DE MEMÓRIA

A alocação e a liberação de memória são casos especiais de comandos intermediários dentro do WebAlgo. Cada um tem o seu tipo de comando, identificado pelo atributo *tipoComando*, além do seu próprio construtor e um método especializado.

Alocação de memória é um comando escrito na forma de uma atribuição em que o ponteiro em questão recebe da função *malloc* o endereço de memória alocado. Não há, no entanto, nenhuma diferença na forma como o analisador sintático reconhece o comando, de forma que se trata apenas de uma atribuição de uma função para a uma variável, o que vai gerar um nodo do tipo **NodoExp** do tipo variável com a função reconhecida como uma expressão de atribuição.

Variáveis que recebem de uma expressão de atribuição são tratadas no método *trata_var*. No caso da alocação de memória, ao executar esse método, a operação é

delegada para uma nova instância da classe **Intermediario** conforme ilustrado no trecho de código contido no Algoritmo 11.

Algoritmo 11 – Criação do comando intermediário de alocação de memória

```
if (ExpAtrib.tipoNodo == NodoExp.chamadafuncao
      && ExpAtrib.nome_func.equals("malloc"))
{

      if (this.tipoRetorno == Constantes.Tp_Ret_Memoria){
            seta_erro(mErro
            ,"Não é possível atribuir um endereço de memória para uma
      variável do tipo " + this.nodoVarResolvida.getStringTipo() + " na
      linha " + this.linhaCodigoExp + "." + " É necessário realizar uma
      operação de casting.");
            return null;
      }
      else {
            if (this.tipoRetorno != this.nodoVarResolvida.getTipo()){
                  seta_erro(mErro
                  ,"Esperava conversão para " +
            this.nodoVarResolvida.getStringTipo() + " na alocação de
            memória realizada " + " na linha " + this.linhaCodigoExp + ".
            Encontrou o tipo " +
            Constantes.toStringConstante(this.tipoRetorno) + ".");
                  return null;
            }
      }

      PNodoExp p = new PNodoExp(); p.nE = ExpAtrib.filhoE;
      Intermediario comandoAlocacao =
Intermediario.CriaComandoAlocacaoMemoria(this.nodoVarResolvida.screen
                                          , this.aSinta, 0
                                          , this.nodoVarResolvida, p);
      comandoAlocacao.executa(mErro);
      return null;
}
```

Fonte: Próprio autor

O método especializado *trataAlocacaoMemoria* se encarrega de realizar todas as validações necessárias e instanciar a variável que será apontada pelo ponteiro em questão para depois alocá-la na memória através da chamada do método de alocação da classe **Heap**.

A “desalocação” funciona de forma similar, mas chamando o método de liberação de memória passando a instância de **OTipo** que deve ser retirada da *heap*. O comando intermediário que realiza a liberação de memória também não é criado diretamente a partir da análise sintática. A exemplo do que ocorre com a alocação ele é criado dentro dos métodos de resoluções de expressões da **NodoExp**. Neste caso,

como o uso da função *free* é uma chamada de função simples a criação do novo comando ocorre no método *trata_chama_func* usando a lógica apresentada no Algoritmo 12.

Algoritmo 12 – Criação do comando intermediário de liberação de memória

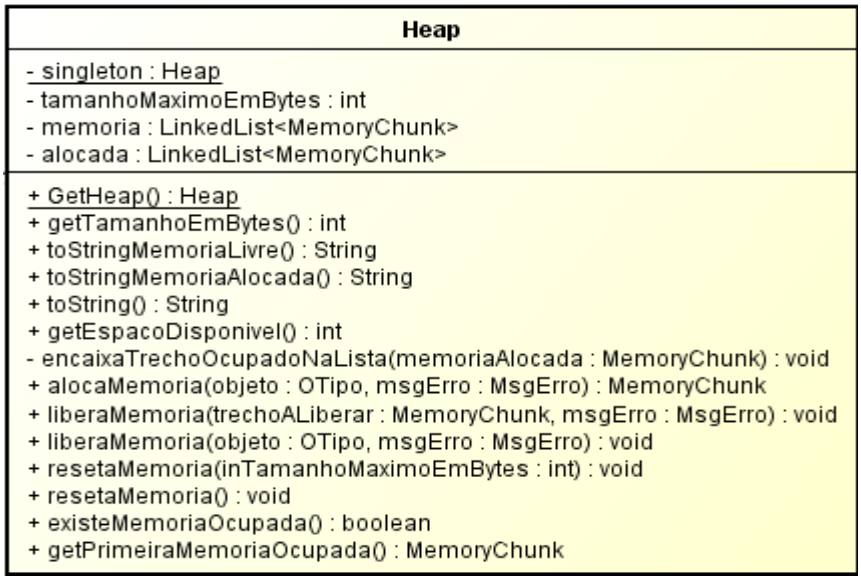
```
if (this.nome_func.equals("free")){

      Intermediario comandoDesalocacao =
Intermediario.CriaComandoDesalocacaoMemoria(this.filhoE.nodoVar.screen
                                          , this.aSinta
                                          , this.linhaCodigoExp
                                          , this.filhoE.nodoVar
                                          );
      comandoDesalocacao.executa(mErro);
      return;
}
```

Fonte: Próprio autor

O funcionamento interno das duas classes que compõem o módulo de memória dinâmica, **Heap** e **MemoryChunk** é basicamente idêntico à descrição da etapa de planejamento, conforme explicado na Seção 5.1.3. As alterações realizadas foram de conveniência, e não de alteração de comportamento.

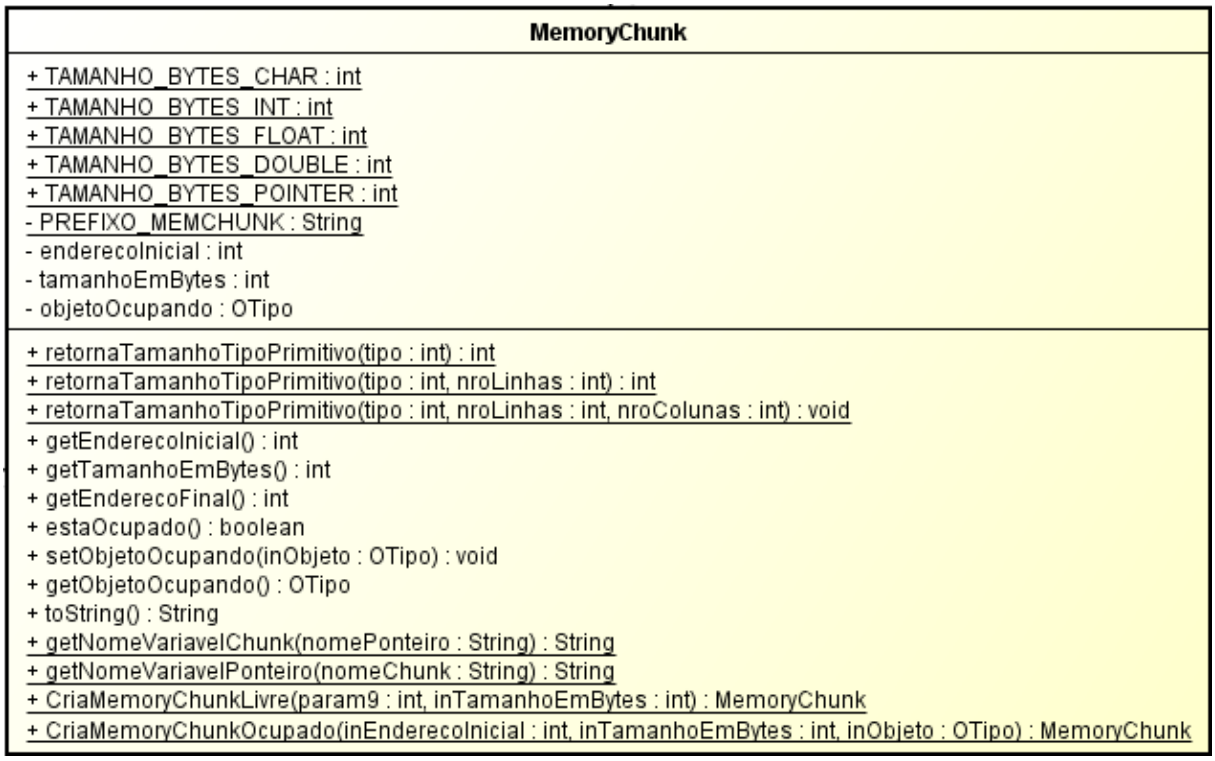
As Figuras 43 e 44 mostram a estrutura final das classes **Heap** e **MemoryChunk**, respectivamente. O apêndice E apresenta os diagramas de sequência atualizados para alocação e “desalocação” de memória.

Figura 43 - Estrutura da classe Heap



Fonte: Próprio autor

Figura 44 - Estrutura da classe MemoryChunk



Fonte: Próprio autor

7 DESCRIÇÃO DE TESTES REALIZADOS

Aqui estão descritos os casos de testes mais importantes utilizados para validação das novas funcionalidades desenvolvidas. A sua importância se deve à sua complexidade e esta, por sua vez, à heterogeneidade das estruturas utilizadas pelo programa que permite que uma ampla gama de novas funcionalidades seja validada.

Foram realizados diversos testes específicos cujo objetivo era aferir o funcionamento de uma construção sintática ou semântica sobretudo no início do projeto e para isolar erros ocorridos durante um dos testes mais importantes. Entretanto, nenhum destes está descrito aqui por questão de praticidade e no intuito de maximizar a utilidade desta seção.

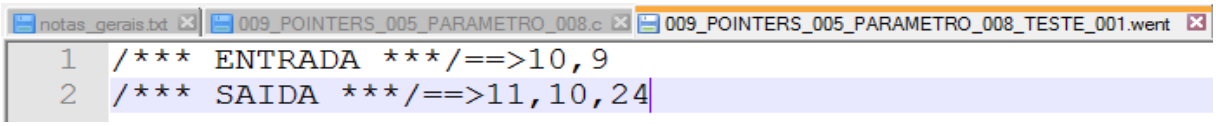
Não há nos testes aqui descritos nenhuma preocupação quanto à sua utilidade como um programa ou mesmo exercício didático real. Alguns deles podem, de fato, ter alguma utilidade, mas isso foi imaterial para os propósitos de validação desse projeto. Cada um dos casos foi montado apenas com o intuito de aferir o funcionamento do que foi desenvolvido.

7.1 TESTE DE OPERAÇÕES COM PONTEIROS

O algoritmo desenvolvido para este teste faz uso dos dois operadores unários aplicáveis sobre ponteiros - endereço e “indireção” - e de expressões que podem ser atribuídas aos mesmos. Além disso, também valida a passagem de ponteiros por parâmetro. Seu código está disponível no apêndice F.

Este é um exemplo de um teste desenvolvido apenas para validação de funcionalidades e que não tem utilidade real. Para ilustrar o seu funcionamento a Figura 45 apresenta um caso de testes desenvolvido para ele.

Figura 45 - Caso de teste para o algoritmo de Teste de Operações com Ponteiros



Fonte: Próprio autor

7.2 TESTE DE VETOR DE STRUCTS

Este algoritmo, disponível para consulta no apêndice G, valida o funcionamento de um vetor de *structs* incluindo o seu uso como parâmetro para uma função. As leituras dos valores são realizadas diretamente com o uso de campos dentro dos elementos do vetor.

7.3 TESTE DE LISTA ENCADEADA

Este algoritmo que lê e ordena uma lista encadeada criada a partir de uma entrada de dados inteiros testa os dois operadores de ponteiros, a criação de uma *struct* contendo auto referência, ou seja, um ponteiro que aponta uma instância da própria *struct*, o que é indispensável para a implementação de listas encadeadas.

Também utiliza a alocação dinâmica de memória e a liberação que ocorre no final do programa, liberando cada nodo um a um partindo do final e terminando no início da lista. E além disso, força o WebAlgo a trabalhar com a referência a um ponteiro nulo através da palavra reservada *NULL*.

Está disponível para consulta no apêndice H.

7.4 NOTA SOBRE TESTES DE REGRESSÃO

Além dos algoritmos aqui descritos também foi utilizada toda a bateria de testes já existentes no banco de dados do WebAlgo e desenvolvida pelo professor orientador, com o objetivo de garantir uma versão estável do WebAlgo sem *bugs* observáveis nas funcionalidades já existentes.

Estes foram, portanto, testes de regressão utilizados durante o desenvolvimento. Uma funcionalidade já existente na versão antiga do programa cuja finalidade é a mesma do testador automático descrito na Seção 6.1 foi utilizada para execução dos testes de regressão. A diferença entre esta e o testador automático é que o último pode ser utilizado sem a necessidade de *login* ou mesmo de acesso à interface gráfica. Além disso, também permite a execução de testes sem a necessidade de cadastrar um programa no banco de dados.

8 CONCLUSÕES

O resultado do projeto aqui apresentado, considerando a etapa de concepção, deve ser considerado positivo pela aplicabilidade das estruturas de dados pensadas para suportar as funcionalidades desenvolvidas e, de forma mais prática, pela entrega de uma nova versão estável do WebAlgo com todos os requisitos implementados e testados.

Em sua versão atual o compilador/interpretador possibilita o uso do programa para outras disciplinas além de programação básica em C. Já é possível o seu uso em Estruturas de Dados, por exemplo, embora por falta de tempo hábil não tenha sido possível realizar a ampla gama de testes necessária para garantir o seu funcionamento para todas as necessidades inerentes ao estudo da disciplina.

8.1 SÍNTESE

O amplo referencial teórico apresentado neste trabalho procura contextualizar o projeto de implementação de forma que as ferramentas analíticas e estruturas necessárias para o desenvolvimento de um compilador/interpretador em C estejam devidamente representadas. Embora muito resumido em alguns aspectos, pois o foco foi direcionado para as necessidades de desenvolvimento levantadas, também foi parcialmente pensado para dar suporte ao que o WebAlgo já faz. Esse estudo também pode ser estendido consultando a bibliografia em (DORNELES, JUNIOR, & ADAMI, 2011) e (WEBER, 2016). Também servirão bem a título de trabalhos anteriores.

Também é de destaque essencial o estudo realizado sobre o gerenciamento de memória em termos de como ele é implementado em sistemas operacionais. O suporte fornecido por essa pesquisa foi de suma importância para a estruturação do simulador de gerenciamento. O fato de toda a implementação ser codificada em Java complementa a justificativa das diferentes seções do referencial teórico.

O trabalho de efetivamente desenvolver as alterações previstas não seguiu uma metodologia ou processo formal. Com mais tempo disponível seria possível desenvolver soluções melhores pois a forma como o WebAlgo trata as estruturas intermediárias poderia ter sido alterada em suas bases, criando assim um código que possibilitasse maior manutenibilidade.

Isso se deve em parte ao fato de que as estruturas desenvolvidas em análise foram pensadas para implementação de novas funcionalidades e não alterações estruturais. Além disso, haveriam benefícios na aplicação padrões de projetos adequados fossem aplicados.

Feitas essas considerações verifica-se, conforme apresentado no Capítulo 5, que o desenvolvimento, embora informal em certa medida, foi caracterizado por alterações incrementais orientadas a testes, como uma metodologia *ad hoc* de melhoria contínua.

A descrição do código implementado, apresentada no Capítulo 6, procura apresentar os principais aspectos do que foi efetivamente codificado, mas sem descrições exaustivas. As subseções estão ordenadas da forma que foi definida no planejamento, mas algumas questões circunstanciais decorridas do desenvolvimento fizeram com que elas não estivessem em ordem cronológica de implementação.

Basicamente o que foi feito por último, por questão de praticidade, foram vetores de *structs* e as tratativas para *structs* auto referenciáveis, isto é, que possuem um ou mais ponteiros apontando para uma instância da própria *struct*. Em ambos os casos foram necessárias novas estruturas de dados para comportá-las. Esses são os únicos desenvolvimentos realizados após a implementação de simulador de gerenciamento de memória.

Alguns dos testes aplicados mais importantes receberam um lugar dentro da Capítulo 7 para ilustrar as funcionalidades do C que passam a fazer parte do conjunto reconhecido pelo WebAlgo. Seu objetivo também é o de delimitar até que ponto o projeto conseguiu avançar para tratar todos os casos possíveis.

8.2 CONTRIBUIÇÕES DO TRABALHO

Está claro para os envolvidos neste projeto, tanto autor quanto orientador e membros da banca que alunos de programação se beneficiam muito mais da prática do que da teoria aprendida em livros. Foi comprovado após a aplicação bem-sucedida do uso do WebAlgo e agora, mais uma vez, corroborado pela sua expansão e consequente ganho durante as atividades didáticas.

A possibilidade de escrever código dentro desta ferramenta didática usando a maior parte do poder da linguagem C, que incidentemente também representa as suas particularidades mais notáveis em comparação com outras das linguagens mais

utilizadas, deve contribuir para que os alunos aprendam na prática a programação com ponteiros, *structs* e alocação dinâmica de memória.

Porém, o projeto possui limitações em alguns pontos da implementação por não possibilitar o uso da totalidade das funcionalidades de um compilador, ou da própria linguagem ou mesmo falha em simular por completo o gerenciamento de memória em algumas situações específicas. Na medida do possível, todas essas limitações estão descritas nos Capítulos 4 e 6. Eventualmente poderão haver outras que não foram contempladas dentro dos casos de testes.

8.3 TRABALHOS FUTUROS

A principal proposta para o desenvolvimento de trabalhos futuros, proveniente de ideias levantadas pelo orientador deste trabalho e outros alunos do curso de Computação, foca no desenvolvimento de funcionalidades didáticas que ajudem o aluno a compreender o funcionamento conceitual das estruturas envolvidas na execução do código fonte, preferencialmente através do uso de ferramentas gráficas.

Expressões em formato de árvore poderiam ser expressas graficamente, mostrando o valor de cada nodo, ou mesmo calculado para cada expressão filha em tempo de execução. O estado da memória *heap* também poderia ser visualizado graficamente, permitindo ao usuário uma depuração muito mais eficiente.

Além disso também existem as limitações incluídas neste projeto, muitas das quais estão descritas nas análises do Capítulo 4 ou nas descrições das funcionalidades no Capítulo 6. Como exemplo pode-se citar a possibilidade do desenvolvimento de um simulador na memória *stack*, conforme descrito no final da Seção 4.1, ou então de vetores com tamanhos variáveis para retirar a limitação descrita na Seção 4.3.1.

Por mais abrangente que as implementações realizadas neste projeto possam ser, e o objetivo era de que fossem tão abrangentes e genéricas quanto possível, certamente haverá mais de um aspecto do C que não está implementado, ou funcionalidades do simulador que podem ser adaptadas para emular de forma mais fiel o funcionamento do C em baixo nível.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, A. V., ULLMAN, J. D., & SETHI, R. (2008). *Compiladores: princípios, técnicas e ferramentas.* São Paulo: Pearson Addison-Wesley.

Appel, A. W. (1999). *Modern Compiler Implementation in Java.* Cambridge University Press.

Bartlett, J. (2003). Programming from the Ground Up. (D. Bruno Jr., Ed.)

Bartlett, J. (16 de November de 2004). Inside Memory Management. *developerWorks*. Acesso em 07 de Abril de 2017, disponível em https://www.ibm.com/developerworks/linux/library/l-memory/

Deitel, H. M., Deitel, P. J., & Choffnes, D. R. (2005). *Sistemas Operacionais* (terceira edição ed.). São Paulo: Pearson Pretince Hall.

DEITEL, P., & DEITEL, H. (2011). C: como programar. (6 ed.).

DORNELES, R. V., JUNIOR, D. P., & ADAMI, A. G. (2011). AlgoWeb: a web-based environment for learning introductory programming. *Renote, v. 9*.

ISO/IEC. (2011). *INTERNATIONAL STANDARD - Programming Languages - C.* unspecified: Commitee Draft.

Lanhellas, R. (s.d.). Diferença entre ArrayList, Vector e LinkedList em Java. *DevMedia*. Fonte: http://www.devmedia.com.br/diferenca-entre-arraylist-vector-e-linkedlist-em-java/29162

Mak, R. (1996). *Writing Compilers and Interpreters: An Applied Aproach Using C++.* United States of America (USA): Wiley Computer Publishing.

Medeiros, E. (2004). *Desenvolvendo Software com UML 2.0: definitivo.* São Paulo: Pearson Makron Books.

ORACLE Corporation. (s.d.). *HashMap (Java Platform SE 7) - Oracle Help Center.* ORACLE Docs. Acesso em 30 de 05 de 2017, disponível em https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

ORACLE Corporation. (s.d.). LinkedList (Java Platform SE 7) - Oracle Help Center. *ORACLE Docs*. Acesso em 30 de 05 de 2017, disponível em https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html

ORACLE Corporation. (s.d.). The Java Virtual Machine Specification. *Oracle Docs*. Acesso em Abril de 2017, disponível em https://docs.oracle.com/javase/specs/jvms/se7/html/

ORACLE Corporation. (s.d.). Understanding Memory Management. *Oracle Docs*. Acesso em Março de 2017, disponível em https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html

QUESTIONABLE UTILITY COMPANY. (2012). *ANSI C Yacc Grammar.* Acesso em 26 de Maio de 2017, disponível em http://www.quut.com/c/ANSI-C-grammar-y.html#type_specifier

Summit, S. (s.d.). *C-Faq.* Acesso em 07 de 12 de 2017, disponível em http://c-faq.com/

TANENBAUM, A. S. (2007). *Modern Operating Systems* (3. ed. ed.). United States of America (USA): Pearson.

WEBER, G. S. (2016). *Desenvolvimento de um Compilador de Português Estruturado para a JVM no Portal de Algoritmos da UCS.* Caxias do Sul.

APÊNDICE A

ALOCADOR ALTERNATIVO DE MEMÓRIA EM C (Bartlett, Inside Memory Management, 2004)

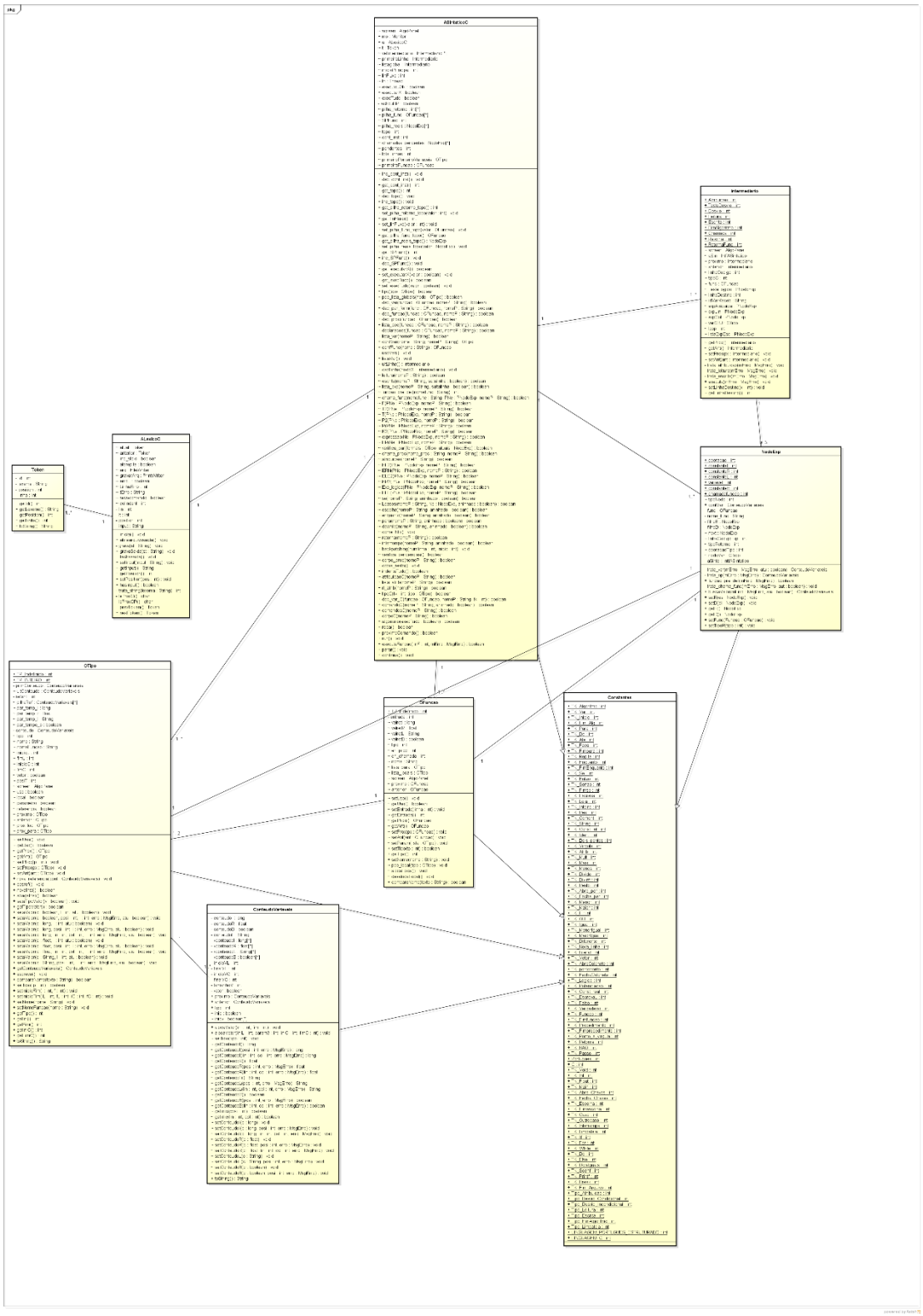
```c
/* Include the sbrk function */
#include <unistd.h>

int has_initialized = 0;
void *managed_memory_start;
void *last_valid_address;

void malloc_init()
{
      /* grab the last valid address from the OS */
      last_valid_address = sbrk(0);

      /* we don't have any memory to manage yet, so
       *just set the beginning to be last_valid_address
       */
      managed_memory_start = last_valid_address;

      /* Okay, we're initialized and ready to go */
      has_initialized = 1;
}

struct mem_control_block {
 int is_available;
 int size;
};

void free(void *firstbyte) {
      struct mem_control_block *mcb;

      /* Backup from the given pointer to find the
       * mem_control_block
       */
      mcb = firstbyte - sizeof(struct mem_control_block);

      /* Mark the block as being available */
      mcb->is_available = 1;

      /* That's It!  We're done. */
      return;
}

void *malloc(long numbytes) {
      /* Holds where we are looking in memory */
      void *current_location;

      /* This is the same as current_location, but cast to a
       * memory_control_block
       */
      struct mem_control_block *current_location_mcb;
```

```
/* This is the memory location we will return.  It will
 * be set to 0 until we find something suitable
 */
 void *memory_location;

/* Initialize if we haven't already done so */
if(! has_initialized)  {
      malloc_init();
}

/* The memory we search for has to include the memory
 * control block, but the user of malloc doesn't need
 * to know this, so we'll just add it in for them.
 */
numbytes = numbytes + sizeof(struct mem_control_block);

/* Set memory_location to 0 until we find a suitable
 * location
 */
memory_location = 0;

/* Begin searching at the start of managed memory */
current_location = managed_memory_start;

/* Keep going until we have searched all allocated space */
while(current_location != last_valid_address)
{
      /* current_location and current_location_mcb point
       * to the same address.  However, current_location_mcb
       * is of the correct type so we can use it as a struct.
       * current_location is a void pointer so we can use it
       * to calculate addresses.
       */
      current_location_mcb =
            (struct mem_control_block *)current_location;

      if(current_location_mcb->is_available)
      {
            if(current_location_mcb->size >= numbytes)
            {
                  /* Woohoo!  We've found an open,
                   * appropriately-size location.
                   */

                  /* It is no longer available */
                  current_location_mcb->is_available = 0;

                  /* We own it */
                  memory_location = current_location;

                  /* Leave the loop */
                  break;
            }
      }
```

```
            /* If we made it here, it's because the Current memory
             * block not suitable, move to the next one
             */
            current_location = current_location +
                  current_location_mcb->size;
      }

      /* If we still don't have a valid location, we'll
       * have to ask the operating system for more memory
       */
      if(! memory_location)
      {
            /* Move the program break numbytes further */
            sbrk(numbytes);

            /* The new memory will be where the last valid
             * address left off
             */
            memory_location = last_valid_address;

            /* We'll move the last valid address forward
             * numbytes
             */
            last_valid_address = last_valid_address + numbytes;

            /* We need to initialize the mem_control_block */
            current_location_mcb = memory_location;
            current_location_mcb->is_available = 0;
            current_location_mcb->size = numbytes;
      }

      /* Now, no matter what (well, except for error conditions),
       * memory_location has the address of the memory, including
       * the mem_control_block
       */

      /* Move the pointer past the mem_control_block */
      memory_location = memory_location + sizeof(struct mem_control_block);

      /* Return the pointer */
      return memory_location;
}
```

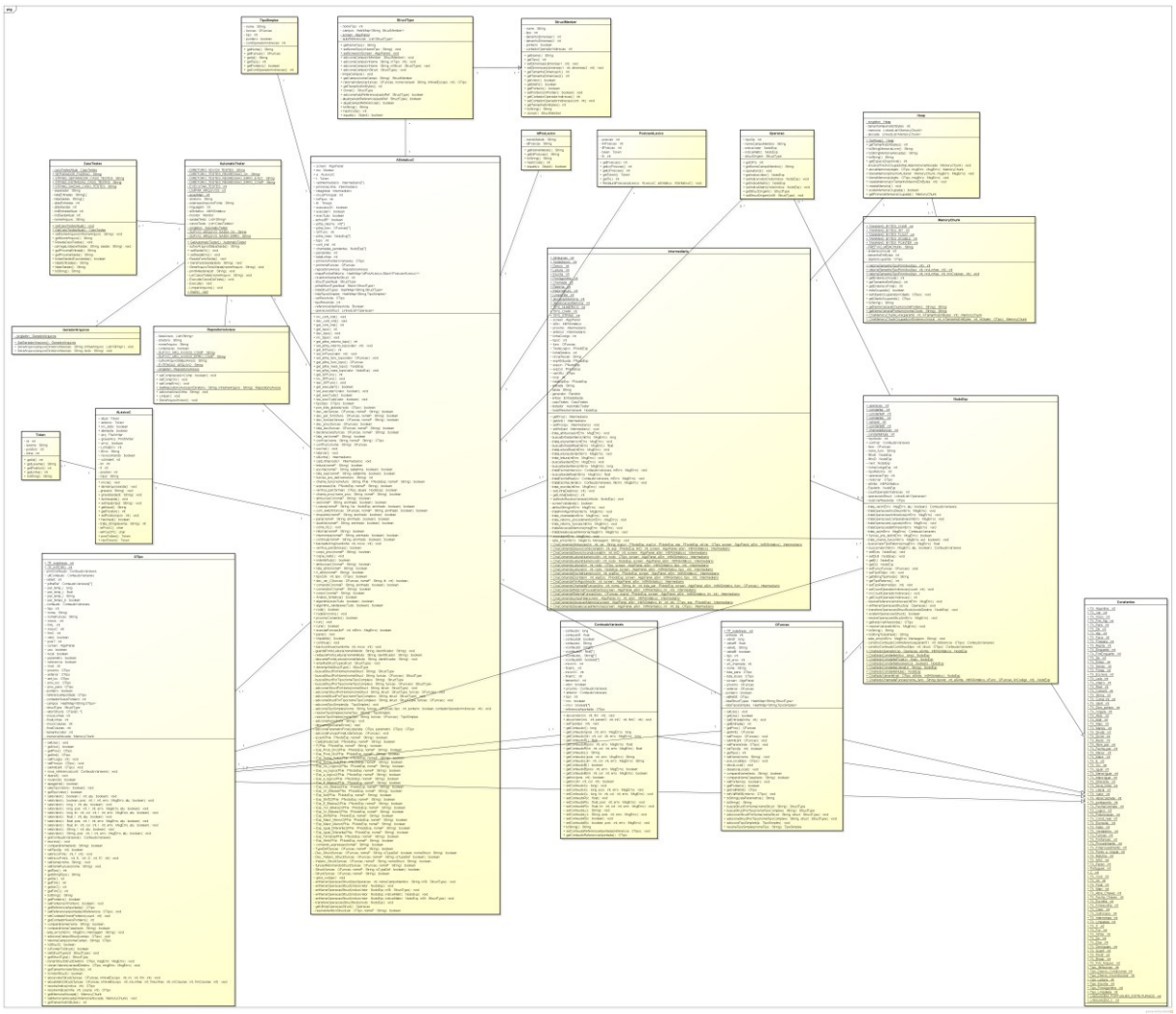
APÊNDICE B

DIAGRAMA DE CLASSES COMPLETO DO COMPILADOR/INTERPRETADOR DO WEBALGO - ANTERIOR AO PROJETO



APÊNDICE C

DIAGRAMA DE CLASSES COMPLETO DO COMPILADOR/INTERPRETADOR DO WEBALGO – FINAL DO PROJETO



APÊNDICE D

MÉTODO *resolveOperacoesStruct*

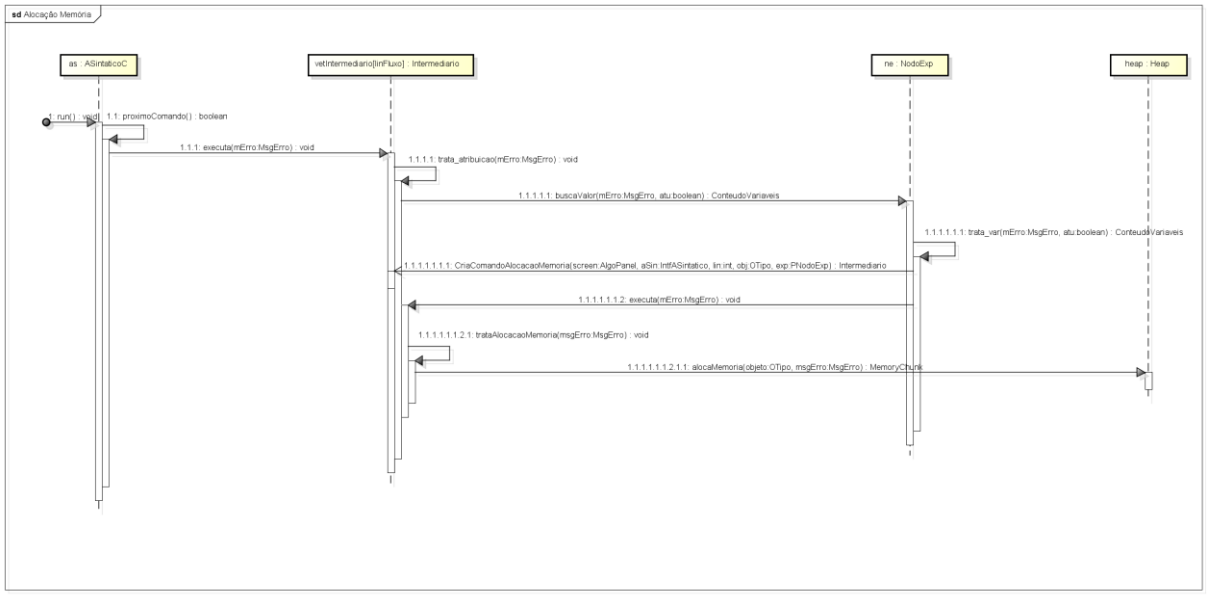
```
private void resolveOperacoesStuct (MsgErro mErro) throws Throwable{
   if (!this.existemOperacoesStruct())
      return;

   Operacao op;
   String nomeCampoMembro;
   //Itera sobre todas as referências não resolvidas
   for (int i = 0; i < this.operacoesStruct.size(); i++){
      op = this.operacoesStruct.get(i);
      if (op.tipoOP != Constantes.TK_AbreColchete
         && op.tipoOP != Constantes.Tk_Ponto
         && op.tipoOP != Constantes.Tk_Flecha){

         seta_erro(mErro,"Operador de struct " +
Constantes.toStringConstante(op.tipoOP) + " inválido na linha "
                                        + this.linhaCodigoExp + ".");
      }

      //resolve elemento de vetor ou matriz
      if (op.tipoOP == Constantes.TK_AbreColchete){
         NodoExp nodoLinha = op.getIndiceVetor();
         if (nodoLinha == null){
            seta_erro( mErro
             , "Erro de programação: Referência a elemento de vetor sem
expressão definida para o índice na linha " + this.linhaCodigoExp + ".");
         }

         NodoExp nodoColuna = op.getIndiceMatriz();
         if (this.nodoVarResolvida.IsVetorStruct()){
            ConteudoVariaveis cvLinha = nodoLinha.buscaValor(mErro
                                                    , false);

            if (nodoColuna != null){
               ConteudoVariaveis cvColuna = nodoColuna.buscaValor(mErro
                                                          , false);
               this.nodoVarResolvida =
this.nodoVarResolvida.resolveIndice( (int) cvLinha.getConteudoI()
                                  , (int) cvColuna.getConteudoI() );
            }
            else {
               this.nodoVarResolvida =
this.nodoVarResolvida.resolveIndice((int) cvLinha.getConteudoI());
            }
         }
```

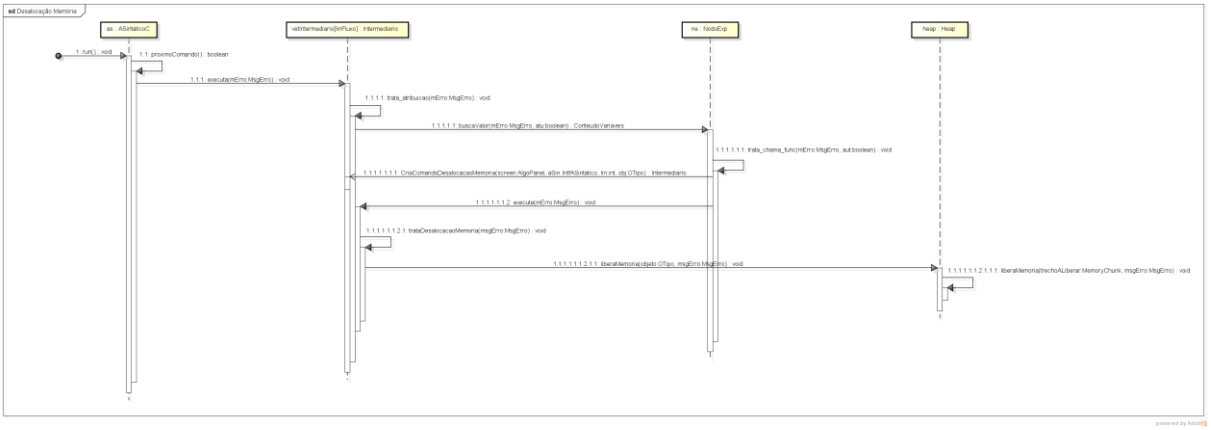
```
          else {
            //se não é última operação então deve ser um vetor de structs
            if ( i < (this.operacoesStruct.size() - 1) ){
               seta_erro(mErro
               , "Erro de programação: Referência a elemento de vetor sem
expressão definida para o índice na linha " + this.linhaCodigoExp + ".");
            }

            this.filhoE = nodoLinha;
            if (nodoColuna != null){
                this.filhoD = nodoColuna;
            }
          }
        }
        else {
           //Resolve membro de struct
           nomeCampoMembro = op.getNomeCampoMembro();

           if (op.tipoOP == Constantes.Tk_Flecha){
              if (!this.nodoVarResolvida.IsPointerToStruct()){
                 seta_erro(mErro
                 ,"Operador '->' aplicado sobre uma variável que não é
ponteiro para struct na linha " + this.linhaCodigoExp + ".");
              }

              if (nomeCampoMembro == null){
                 seta_erro(mErro
                 ,"Operador '->' aplicado sem um campo membro definido na
linha " + this.linhaCodigoExp + ".");
              }

              OTipo pointer = this.nodoVarResolvida;
              this.nodoVarResolvida =
this.nodoVarResolvida.getReferenciaApontada();
              if (this.nodoVarResolvida == null){
                 seta_erro(mErro
                 ,"Ponteiro '" + pointer.nome + "' não aponta para uma
variável ou localização de memória. " + "Impossível resolver o operador '-
>' na linha " + this.linhaCodigoExp  + ".");
              }

              if (!this.nodoVarResolvida.IsStruct()){
                 seta_erro(mErro
                 , "Erro de programação: Ponteiro '" + pointer.nome + "'
não aponta para uma struct na linha " + this.linhaCodigoExp + "."
                 + " Está apontando para a variável " +
this.nodoVarResolvida.nome + ".");
              }
           }
```

```
       else if (op.tipoOP == Constantes.Tk_Ponto){
         if (!this.nodoVarResolvida.IsStruct()){
            seta_erro(mErro
            , "Erro de programação: Variável '" +
this.nodoVarResolvida.nome + "' não é uma struct. Ocorrido na linha " +
this.linhaCodigoExp + ".");
          }
        }

        OTipo campoMembro =
this.nodoVarResolvida.retornaCampo(nomeCampoMembro);
        if (campoMembro == null){
           seta_erro(mErro
           ,"Erro de programação: Struct '" + this.nodoVarResolvida.nome
+ "' não contém o membro '" + nomeCampoMembro + "'." + " Ocorrido na linha
" + this.linhaCodigoExp + ".");
         }

         this.nodoVarResolvida = campoMembro;
         this.tipoRetorno = this.nodoVarResolvida.getTipo();
       }
   }
}
```

APÊNDICE E

DIAGRAMAS DE SEQUÊNCIA DE GERENCIAMENTO DE MEMÓRIA





APÊNDICE F

TESTE DE OPERAÇÕES COM PONTEIROS

```c
#include <stdio.h>
int s;

int * func() {
   int i; int *r;
   for ( i = 0; i <= 10; i++) {
      if (i == s){
         r = &s; break;
      }
   }
   (*r)++;
   return r;
}

void func2(int * r){
   int e = s - 1;
   *r = e;
}

void func3 (int * w){
   *w += (s + 5);
}

int main () {
   scanf("%d",&s);
   int i;
   scanf("%d",&i);

   int *p;
   p = func();
   printf("%d",*p);

   func2(p);
   printf("%d",*p);
   func3(&i);
   printf("%d",i);

   return 0;
}
```

APÊNDICE G

TESTE DE VETOR DE STRUCTS

```c
#include <stdio.h>

struct import {
       int id;
       float val;
};

typedef struct import tp_import;

float getAverage(tp_import arr[5], int size) {
   int i;
   float avg;
   float sum = 0;

   for (i = 0; i < size; i++) {
      sum += arr[i].val;
   }

   avg = sum / size;
   return avg;
}

int main () {
   tp_import balance[5];
   float avg;

   int i = 0;
   for (i = 0;i < 5; i++){
       scanf("%d",&balance[i].id);
       scanf("%f",&balance[i].val);
   }

   avg = getAverage( balance, 5 );
   printf( "%f", avg );
   return 0;
}
```

APÊNDICE H

TESTE DE LISTA ENCADEADA

```
#include <stdio.h>
#include <stdlib.h>

struct nodo {
	int elemento;
	struct nodo *anterior;
	struct nodo *proximo;
};

struct nodo *inicio;
struct nodo *final;

int *nrocomp;
int *nrotrocas;

void lerEntrada () {
	struct nodo *atual;
	struct nodo *prox;

	int i;
	int e;
	for (i = 0; i < 10; i++){

		scanf("%d",&e);

		if (atual == NULL){
			atual = (struct nodo *) malloc(sizeof(struct nodo));
			atual->elemento = e;
			inicio = atual;
			final = atual;
		}
		else {
			prox = (struct nodo *) malloc(sizeof(struct nodo));
			prox -> elemento = e;
			prox -> anterior = atual;

			atual -> proximo = prox;
			prox -> anterior = atual;
			final = prox;
			atual = prox;
		}
	}
}
```

```
void escreveSaida(){
	struct nodo *atual;
	atual = inicio;

	while (atual != NULL){
		printf("%d",atual->elemento);
		atual = atual -> proximo;
	}
}

void bubblesort (int *nrocomp,int *nrotrocas) {
	struct nodo *atual;
	struct nodo *troca;
	int aux;

	atual = inicio;

	while (atual != NULL){

		troca = atual -> proximo;
		while (troca != NULL){

			*nrocomp = *nrocomp + 1;
			if (atual->elemento > troca->elemento){
				aux = atual->elemento;
				atual->elemento = troca ->elemento;
				troca ->elemento = aux;

				*nrotrocas = *nrotrocas + 1;
			}
			troca = troca -> proximo;
		}
		atual = atual -> proximo;
	}
}

void liberaMemoria(){
	struct nodo *atual;
	struct nodo * liberado;
	atual = final;

	while (atual != NULL){
		liberado = atual;
		atual = atual -> anterior;
		free(liberado);
	}
}

int main () {
	lerEntrada();
	int nrocomp = 0; int nrotrocas = 0;
	bubblesort(&nrocomp,&nrotrocas);
	escreveSaida(); liberaMemoria();
    return 0;
}
```