UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

ISMAEL STANGHERLINI

CUIA: Uma Ferramenta para a Obtenção de Informações de Variáveis em Códigos C

Trabalho de Conclusão apresentado como
requisito parcial para a obtenção do grau de
Bacharel em Ciência da Computação

Prof. Dr. Nicolas Maillard
Orientador

Porto Alegre, junho de 2010

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Stangherlini, Ismael

CUIA: Uma Ferramenta para a Obtenção de Informações de Variáveis em Códigos C / Ismael Stangherlini. – Porto Alegre, 2010.

67 f.: il.

Trabalho de Conclusão (mestrado) – Universidade Federal do Rio Grande do Sul. Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR–RS, 2010. Orientador: Nicolas Maillard.

1. C. 2. NUMA. 3. OpenMP. I. Maillard, Nicolas. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitora de Graduação: Profa. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do Curso: Prof. João César Netto
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 5

LISTA DE FIGURAS 6

LISTA DE LISTAGENS 7

RESUMO 8

ABSTRACT 9

1 INTRODUÇÃO 10
1.1 Contextualização Científica 10
1.1.1 As Arquiteturas *NUMA* 11
1.1.2 O Projeto *MApp* 12
1.2 Objetivo 12
1.3 Organização do Texto 13

2 AS INFORMAÇÕES RECUPERADAS E O FORMATO DA SAíDA . . . 14
2.1 As Informações Recuperadas 14
2.2 O Formato da Saída 15

3 *LEX* E *YACC*: AS FERRAMENTAS AUXILIARES UTILIZADAS 18
3.1 A Análise Léxica e a Análise Sintática 18
3.2 As Ferramentas Existentes 19
3.3 As Ferramentas Escolhidas: *Lex* e *Yacc* 20
3.4 A Sintaxe dos Pseudocódigos 21

4 AS ESTRUTURAS DE DADOS UTILIZADAS 23
4.1 As Funções Utilizadas 23
4.1.1 **cria_entrada** ( ) 24
4.1.2 **insere_entrada** (tabela *t*, entrada *e*) 24
4.1.3 **retira_entrada** (tabela *t*, entrada *e*) 24
4.1.4 **imprime_tabela** (tabela *t*) 24
4.1.5 **empilha** (pilha *p* , descritor *d*) 24
4.1.6 **desempilha** (pilha *p*) 25
4.1.7 **topo_pilha** (pilha *p*) 25
4.1.8 **pilha_vazia** (pilha *p*) 25
4.1.9 **coloca_fila** (fila *f* , string *s*) 25
4.1.10 **concatena_elementos_fila** (fila *f*) 25

4.1.11 **concatena_asteriscos** (int *n* , string *s*) 25
4.1.12 **cria_elemento_lista_dimensoes** (int *n*) 25
4.1.13 **cria_elemento_lista_acessos** (int *n* , int *linha* , int *coluna* , string *arquivo*) 25
4.1.14 **insere_lista** (lista *l* , elemento *e*) 25
4.1.15 **existe_elemento_lista** (lista *l* , int *linha* , int *coluna* , string *arquivo*) . . 26

5 A CONSTRUÇÃO DA FERRAMENTA CUIA 27
5.1 A Obtenção das Informações 27
5.1.1 Obtenção do Nome da Variável 29
5.1.2 Obtenção do Tipo da Variável 30
5.1.3 Obtenção do Escopo da Variável 32
5.1.4 Obtenção do Nome do Arquivo em que a Variável é Declarada 36
5.1.5 Obtenção do *Tipo de Array* de uma Variável 39
5.1.6 Obtenção de Informações Extras Relativas a Arrays 43
5.2 O Tratamento das Diretivas de Pré-Processamento 52

6 A VALIDAÇÃO DA FERRAMENTA 55
6.1 Os Benchmarks e a Aplicação de Teste 55
6.2 As Restrições de Uso 56
6.2.1 Os Tamanhos das Dimensões 56
6.2.2 Utilização do *#define* 56
6.2.3 As Estruturas e Enumerções 57

7 CONCLUSÕES 58

REFERÊNCIAS 59

ANEXO A - GRAMÁTICA DA LINGUAGEM C 60

LISTA DE ABREVIATURAS E SIGLAS

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
BNF	Backus-Naur Form
CUIA	Code Under examination to retrieve InformAtions
HPC	High Performance Computing
IDE	Integrated Development Environment
Lex	Lexical Analyzer Generator
NUMA	Non-Uniform Memory Access
MApp	Memory Affinity preprocessor
OpenMP	Open Multi-Processing
Yacc	Yet Another Compiler-Compiler

LISTA DE FIGURAS

Figura 1.1: Diferenciação Entre Acesso Uniforme e Não-Uniforme à Memória . . 11

Figura 2.1: Exemplo de Código-Fonte C Analisado 17
Figura 2.2: Exemplo de Saída da Ferramenta *CUIA* 17

Figura 3.1: Fluxo de Construção da Ferramenta CUIA com o Auxílio das Ferramentas Lex e Yacc 22

Figura 4.1: Entrada de Uma Variável e Seus Respectivos Nomes de Campos Utilizados 24

Figura 5.1: Recuperação de Informações Dividida em Tipos de Produções 28
Figura 5.2: Seqüência de Varredura de Código C com Respectiva Situação da Pilha de Escopos 37
Figura 5.3: Seqüência de Varredura de Código C com Respectiva Situação da Pilha de Arquivos 38
Figura 5.4: Seqüência de Reduções para a Obtenção do Tipo Final de um Array Dinâmico 44
Figura 5.5: Obtenção do Tamanho das Dimensões do Array 47

LISTA DE LISTAGENS

5.1 Exemplo de Código C 28
5.2 Especificação da Ação da Análise Léxica para Casamentos de Identificadores 29
5.3 Ação Semântica para Criação de Nova Entrada na Tabela de Símbolos e Obtenção do Nome da Variável 30
5.4 Exemplos de Declarações de Variáveis na Linguagem C 30
5.5 Ações Semânticas para o Salvamento dos Tokens do Tipo de uma Variável 31
5.6 Ação Semântica para Atualização do Tipo de uma Variável 31
5.7 Inserção do Escopo na Entrada da Tabela de Símbolos 33
5.8 Ações Semânticas para Empilhar o Escopo de uma Função 34
5.9 Ações Semânticas para Desempilhar o Escopo de uma Função 35
5.10 Ações Semânticas para Empilhar o Escopo de um Bloco 36
5.11 Ação Semântica para Desempilhar o Escopo de um Bloco 36
5.12 Inserção do Nome do Arquivo na Entrada da Tabela de Símbolos 39
5.13 Exemplo de Declaração e Uso de Array Dinâmico em C 40
5.14 Exemplo de Declaração e Uso de Array Estático em C 40
5.15 Atualização da Entrada com a Informação *NOT_ARRAY* 41
5.16 Atualização da Entrada com a Informação *STATIC_ARRAY* 42
5.17 Ações Semânticas para a Contagem do Número de Níveis de Ponteiros . . 42
5.18 Atualização da Entrada com a Informação *DYNAMIC_ARRAY* e o Tipo Final 43
5.19 Ações Semânticas para Reconhecimento do Tamanho das Dimensões . . 44
5.20 Exemplos de Declarações de Arrays Estáticos em C 45
5.21 Ações Semânticas para o Cálculo de uma Expressão Constante 45
5.22 Ações Semânticas para a Contagem do Número de Dimensões de um Array 46
5.23 Ações Semânticas para Manipulação de *pilha_fors* 48
5.24 Ações Semânticas para Manipulação de *pilha_fors* 49
5.25 Ações Semânticas para Reconhecimento de Acesso de Escrita 50
5.26 Ações Semânticas para o Reconhecimento de Acesso de Leitura 51
5.27 Exemplo de Utilização da Diretiva *#define* em C 52
5.28 Exemplo de Utilização de Diretivas *#define* Aninhadas em C 53
6.1 Exemplo de Definição de Tamanho de Array com Valor de Identificador, em C 56
6.2 Exemplos de Utilizações Aceitas para a Diretiva *#define* 56
6.3 Exemplo de Utilização da Diretiva *#define* Não Aceita pela Ferramenta . 56

RESUMO

As arquiteturas *NUMA* têm sido amplamente utilizadas como máquinas para computação intensiva de aplicações paralelas, na área de *High Performance Computing*. Tais arquiteturas são caracterizadas pela presença de núcleos de processamento que compartilham diversos níveis de uma memória hierárquica. Para que o desempenho de aplicações utilizadas nessas arquiteturas seja alcançado, é importante que políticas de alocação de memória e de threads sejam utilizadas a fim de garantir a afinidade de memória. O padrão OpenMP, contudo, não possui suporte para as arquiteturas *NUMA*, pecando em tal aspecto. O Projeto *MApp*, por sua vez, surgiu com o objetivo de desenvolver um pré-processador que incorpore políticas de memória *NUMA* diretamente no código-fonte de aplicações desenvolvidas em C com OpenMP.

Este trabalho teve o objetivo de desenvolver uma ferramenta que serviu como o módulo do projeto *MApp* responsável pela extração de informações específicas de variáveis em códigos-fonte C, tais como o tipo de acesso e o escopo de sua utilização. Ferramentas clássicas da área de compiladores, como o Lex e o Yacc, foram utilizadas para esse desenvolvimento, de forma a auxiliar no reconhecimento léxico e sintático. As informações obtidas foram necessárias para que as políticas de memória investigadas pelo Mapp fossem corretamente incorporadas nas aplicações.

Embora tenha sido originalmente concebida para o *MApp*, a ferramenta desenvolvida é independente de contexto e pode ser utilizada para a coleta de informações de quaisquer códigos C, uma vez que essas informações são emitidas em um formato simples e portável na saída.

Palavras-chave: C, NUMA, OpenMP.

CUIA: A Tool for Obtainment of Variables Informations in C Codes

ABSTRACT

NUMA architectures have been widely used as machines to insensive computing of parallel applications in the High Performance Computing area. Such architectures are characterized by the presence of processors that share various levels of a hierarchical memory. To achive performance in applications used in these architectures, it is important to ensure memory affinity with the use of memory policies. The OpenMP standard, however, does not support *NUMA* architectures. The *MApp* project, on the other hand, came up with the aim of developing a preprocessor that incorporates *NUMA* memory policies directly in the source code of applications developed with C and OpenMP.

This study aimed to develop a tool that served as the *MApp* project module responsible for the extraction of specific information about variables in C codes, such as access types and scope. The lexical and syntactic analysis required for this development were covered with the use of two compilers classic tools: Lex and Yacc. This extraction was necessary to correctly incorporate memory policies in the applications.

Although it was originally designed for *MApp*, the developed tool is context-free and can be used to collect information from any C code, since these informations are emited in a simple and portable output format.

Keywords: C, NUMA, OpenMP.

1 INTRODUÇÃO

Este capítulo contém uma introdução ao Trabalho de Conclusão de curso. O tema deste trabalho é *CUIA: Uma Ferramenta para a Obtenção de Informações de Variáveis em Códigos C*. As seções a seguir tratam do contexto científico, dos objetivos almejados e da organização do trabalho.

1.1 Contextualização Científica

Os avanços tecnológicos conquistados nas últimas décadas permitiram a automatização de processos que, anteriormente, eram realizados manualmente pelo homem. Além disso, as fronteiras que limitavam a comunicação foram rompidas de uma forma que revolucionou o modo de agir das pessoas, nos quatro cantos do globo. Um simples apertar de um botão ou o reconhecimento de uma fala podem permitir controlar um meio de transporte ou realizar a comunicação com outra pessoa no outro lado do mundo. Impulsionando também o processo de globalização, a tecnologia computacional permitiu o desenvolvimento de diversas áreas do conhecimento, enriquecendo o nosso *background* científico e descobrindo novas fontes de pesquisa biológica, para a prevenção e o tratamento de doenças.

Dentro do contexto tecnológico, são inúmeros os desenvolvimentos de *software* e *hardware* criados para revolucionar o modo como vivemos, dia após dia. Não é incomum percebermos as mudanças existentes no modo como utilizamos um celular, por exemplo, no decorrer dos últimos dez anos. A idéia original para tal dispositivo era a de apenas suprir um canal de comunicação entre duas pessoas em qualquer canto do globo. Porém, o aprimoramento computacional permitiu a ampliação das funcionalidades presentes nesse dispositivo, como a captação de imagens, o reconhecimento de comandos de voz, a visualização de mapas e, principalmente, a comunicação com a Internet.

No campo científico, são comumente encontradas aplicações que requerem uma elevada capacidade de processamento, já que o volume de dados utilizado nas mesmas é consideravelmente grande. O desenvolvimento da área de HPC[1], por conseguinte, passou a ser realizada com o intuito de ampliar os horizontes científicos e permitir processamentos rápidos de algoritmos que solucionassem tarefas complexas como a previsão do tempo e a dinâmica de fluidos. Nesse contexto, vários núcleos de processamento, por exemplo, foram incorporados em um mesmo sistema, a fim de multiplicar a capacidade computacional e reduzir os custos.

[1] *High Performance Computing*

1.1.1 As Arquiteturas *NUMA*

Para soluções que demandem desempenho, como os problemas de HPC, a disputa pelo acesso à memória por diferentes núcleos de processamento é uma questão a ser enfrentada. Nesse contexto, surgiram as arquiteturas *NUMA*[2], caracterizadas principalmente pela variação significativa no tempo de acesso à memória pelos processadores (SILBERSCHATZ; GALVIN; GAGNE, 2009). Tais arquiteturas são sistemas multi-processados, de forma que os núcleos de processamento são servidos por uma memória compartilhada, a qual é fisicamente distribuída em vários bancos de memória interconectados por uma rede (RIBEIRO et al., 2008). A Figura 1.1 demonstra a diferenciação existente entre acessos uniformes e não-uniformes à memória, típicos de uma arquitetura desse porte.

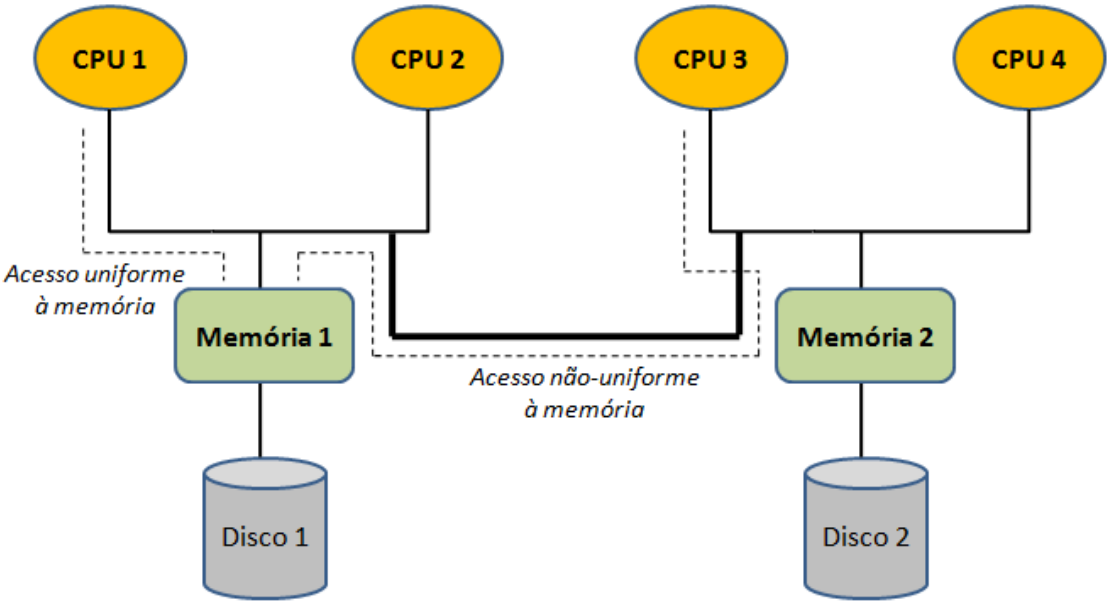

Figura 1.1: Diferenciação Entre Acesso Uniforme e Não-Uniforme à Memória

A interconexão entre os diversos bancos de memória de uma arquitetura *NUMA* gera os diferentes custos de acesso à memória pelos núcleos de processamento. Assim sendo, um acesso realizado por um nodo a sua memória local certamente será mais rápido que o acesso a uma memória que encontra-se ligada a outro nodo. A otimização do desempenho de aplicações paralelas nessas arquiteturas pode ser alcançada, portanto, minimizando-se o número de acessos remotos realizados pelas unidades de processamento.

A afinidade de memória é um conceito que está diretamente relacionado com a redução da distância entre os processos e os dados usados por eles, ou seja, minimização do número de acessos remotos aos dados (MU; TAO; MCKEE, 2003). Para garantir essa afinidade, o sistema operacional provê políticas para gerenciar a alocação de memória e o escalonamento de processos nas arquiteturas *NUMA*. Entretanto, essas políticas não apresentam sempre o desempenho ótimo para todos os tipos de aplicação. Então, nos últimos anos, sistemas operacionais como o Linux e o Solaris têm provido ferramentas em nível de usuário e APIs com chamadas de sistema que permitem aos programadores gerenciar explicitamente a distribuição/alocação de memória e processos das aplicações (RIBEIRO et al., 2008) (CARISSIMI et al., 2007).

Uma das principais vantagens de se utilizar arquiteturas *NUMA* é a possibilidade de

[2] *Non-Uniform Memory Access*

se desenvolver aplicações usando o modelo de programação de memória compartilhada. Nesse modelo de programação, as aplicações paralelas são desenvolvidas considerando-se que a memória é acessada por múltiplos fluxos de execução (RIBEIRO et al., 2008). O OpenMP (OPENMP, 2008), embora seja considerada uma API padrão para o desenvolvimento dessas aplicações nesse modelo, não possui suporte para arquiteturas *NUMA*. Isso significa dizer que o padrão OpenMP, amplamente utilizado na área de HPC, não possui nenhuma otimização de memória para arquiteturas com múltiplos níveis de memória compartilhada.

1.1.2 O Projeto *MApp*

Uma vez que o OpenMP não possui suporte a arquiteturas *NUMA*, desenvolveu-se o Projeto *MApp*[3] (STANGHERLINI et al., 2010), o qual trata-se da implementação de um pré-processador que provê um controle transparente da afinidade de memória em aplicações OpenMP sobre plataformas *NUMA*. Esse controle transparente é realizado através da adição automática, no código-fonte, de chamadas de sistema específicas do Linux, que são responsáveis por um controle mais fino da alocação de memória e das threads nas aplicações.

A transparência do *MApp* reside no fato de que o programador das aplicações paralelas com OpenMP não precisa se preocupar com o controle de memória específico de arquiteturas *NUMA*. O pré-processador desenvolvido, portanto, engloba a função de percorrer o código-fonte da aplicação e extrair informações das variáveis existentes. Tais informações, usadas em conjunto com as características de hardware, são consideradas para a tomada de decisão de quais políticas de memória e chamadas de sistema devem ser aplicadas para *cada* variável. Tendo-se a decisão tomada, o pré-processador *Mapp* modifica o código-fonte original para adicionar tais políticas de memória, com o intuito final de que a afinidade de memória seja alcançada nas arquiteturas *NUMA*.

O *parser* desenvolvido e incorporado ao projeto *MApp* para a extração de informações de variáveis em códigos-fonte denominou-se de *CUIA*[4], e é a ferramenta desenvolvida para este Trabalho de Conclusão.

1.2 Objetivo

O objetivo deste Trabalho de Conclusão é a implementação de uma ferramenta que extraia automaticamente informações específicas de variáveis em códigos C. É importante notar que, dentro do contexto do projeto *MApp*, a ferramenta *CUIA* trata-se de um módulo específico para a extração de informações. Nesse contexto, as informações resultantes são repassadas para os outros módulos do projeto, para que as políticas de memória adequadas sejam escolhidas através de heurísticas específicas.

Embora a *CUIA* seja uma ferramenta especificamente utilizada como o módulo de um projeto maior, suas funcionalidades não se restringem ao *MApp*. Assim sendo, especificou-se uma saída padrão em formato Unicode das informações extraídas, de tal forma que quaisquer ferramentas ou usuários externos possam utilizá-las para outros fins, como a otimização de código, por exemplo.

Um ponto importante a ser considerado é que a ferramenta desenvolvida *não* é um compilador, de forma que os códigos-fonte fornecidos como entrada para a mesma devem ser sintática e semanticamente corretos.

[3] *Memory Affinity preprocessor*

[4] *Code Under examination to retrieve InformAtions*

1.3 Organização do Texto

A organização do texto deste trabalho está organizado da forma explicada a seguir. O capítulo 2 apresenta as informações de variáveis a serem extraídas de códigos C e o formato de saída da ferramenta. O capítulo 3, por sua vez, apresenta alguns conceitos básicos dos processos de análise léxica e sintática e as ferramentas autiliares utilizadas para automatizar a construção desses processos: o *Lex* e o *Yacc*. No capítulo 4, são demonstradas as estruturas de dados utilizadas no trabalho e as respectivas funções para manipulá-las. O capítulo 5 é o cerne da explicação da construção da ferramenta *CUIA*, de forma que a obtenção de cada uma das informações é detalhada ao leitor. Concluido-se o trabalho, o capítulo 6 demonstra a validação da ferramenta, enquanto que o capítulo 7 apresenta as conclusões finais deste trabalho. Por fim, no anexo A, pode-se observar a gramática utilizada para o desenvolvimento da ferramenta.

Ao longo do trabalho, diversas listagens serão demonstradas para explicar o procedimento de construção da ferramenta. Muitas delas são construídas adicionando-se novos comandos à listagens já vistas anteriormente, a fim de facilitar a compreensão das funcionalidades incorporados ao sistema, ao longo do texto. O pseudocódigo utilizado nessas listagens será detalhado no capítulo 3.

2 AS INFORMAÇÕES RECUPERADAS E O FORMATO DA SAÍDA

Este capítulo tem a função de listar cada uma das informações coletadas pela ferramenta *CUIA*, bem como a forma de emissão das mesmas para a saída.

2.1 As Informações Recuperadas

O desenvolvimento de *CUIA* tem como objetivo a recuperação de informações relativas a variáveis presentes em códigos C. Dentro desse contexto, *CUIA* emite na sua saída uma lista com as informações recuperadas, para que as mesmas possam ser consumidas por um usuário ou módulo externo. Embora, como aplicação prática, a ferramenta tenha sido utilizada como um módulo do projeto *MApp*, a sua lista de informações de saída pode ser reaproveitada por quaisquer outras ferramentas que demandem o seu conhecimento. Alguns dos possíveis objetivos para essas ferramentas externas podem ser o alcance de otimização de código e da própria realocação de memória.

Para cada variável encontrada no código, são analisadas e salvas as seguintes informações:

1) **Nome da variável**
2) **Tipo da variável**
3) Indicativo mostrando se a variável é um **array estático, dinâmico ou se não é um array**
4) **Nome do arquivo** em que a variável é declarada
5) **Escopo da variável**, sendo que este engloba três informações:
   (a) **Nome do escopo**, se existir
   (b) **Número da linha** relativa ao início do escopo de bloco, se existir
   (c) **Número da coluna** relativa ao início do escopo de bloco, se existir

Tipicamente, em aplicações paralelas, o acesso à memória é um dos fatores críticos para o alcance de desempenho. Aplicações OpenMP em arquiteturas *NUMA* devem se preocupar, principalmente, com estruturas que utilizam-se de unidades variáveis e relativamente grandes de memória, tais comos os arrays. Muitos dos acessos a essas estruturas são realizadas em laços e comandos do tipo *for*, sendo um dos focos do OpenMP para a paralelização. Nesse caso, para que o preprocessador *MApp* pudesse escolher políticas de alocação de memória adequadas em arquiteturas *NUMA* para aplicações desenvolvidas com essa API, tornou-se necessária a obtenção de algumas informações extras. A ferramenta *CUIA*, portanto, seleciona e armazena outras três informações importantes para todas as variáveis que tenham sido declaradas como do tipo array, sendo elas estáticas ou dinâmicas. Essas informações são listadas abaixo:

1) **Número de dimensões** da variável, para o caso de ser um array estaticamente declarado
2) **Tamanho de cada uma das dimensões** da variável, para o caso de ser também um array estaticamente declarado
3) **Lista com os tipos de acesso** que são realizados pela variável em cada escopo de comandos *for* no código. Neste caso, para cada tipo de acesso, são guardadas as seguintes informações:
   (a) **Tipo de acesso** da variável, podendo este ser de escrita, leitura ou escrita/leitura
   (b) **Números da linha e coluna** relativa ao comando *for* em cujo escopo a variável possui tal acesso
   (c) **Nome do arquivo** relativo ao comando *for* em cujo escopo a variável possui tal acesso

2.2 O Formato da Saída

O formato básico da saída fornecida pela *CUIA* trata-se de um arquivo de texto simples, de forma a facilitar o acesso às informações das variáveis pelo *parser* de uma ferramenta externa, tal como o *MApp*. Nesse caso, em cada linha do arquivo encontram-se todas as informações de uma variável específica, respeitando uma sintaxe bem formada, e que será explicada a seguir.

Dentro de uma mesma linha, o modo utilizado de separação de duas informações consecutivas é a colocação de um, e somente um, caractere *espaço* entre as mesmas. Além disso, todas as informações do tipo *string*, como o nome da variável, são emitidas entre aspas simples. A razão de se utilizar essa metodologia é a de que a informação de tipo de uma variável pode conter *espaços*[1]. Nesse caso, colocando-se uma informação do tipo *string* entre aspas simples evita possíveis ambiguidades e facilita o processo de *parsing* da saída.

A seguir, encontra-se o formato específico para cada informação, na ordem como são encontradas na saída:

1) Nome da variável: ***'nome_variavel'***
2) Tipo de array da variável:
   (a) ***NOT_ARRAY*** para variáveis que não são arrays
   (b) ***STATIC_ARRAY*** para arrays estáticos
   (c) ***DYNAMIC_ARRAY*** para arrays dinâmicos
3) Tipo da variável: ***'tipo_variavel'***
4) Número de dimensões da variável como um valor inteiro (presente somente em arrays estáticos)
5) Tamanho de cada uma das dimensões, separados por *espaço*, sendo valores inteiros (presente somente em arrays estáticos)
6) Tipo de escopo:
   (a) ***GLOBAL_SCOPE*** para variáveis de escopo global
   (b) ***FUNC_SCOPE*** para variáveis de escopo local a uma função
   (c) ***BLOCK_SCOPE*** para variáveis de escopo local a um bloco
7) Nome da função em cujo escopo a variável pertence (presente somente em variáveis de escopo local a uma função) : ***'nome_funcao'***
8) Número da linha de início de bloco em cujo escopo a variável pertence (presente

[1] Exemplo de tipo contendo *espaços*: *static long long int*

somente em variáveis de escopo local a um bloco), sendo um valor inteiro

9) Número da coluna de início de bloco em cujo escopo a variável pertence (presente somente em variáveis de escopo local a um bloco), sendo um valor inteiro
10) Tipo de acesso em comandos *for* para arrays:
    (a) O caractere hifen (**-**) para variáveis sem acesso em comandos *for*, ou que não sejam arrays
    (b) Tipos de acesso, separados por *espaço*, sendo os componentes de um tipo específico colocados entre parênteses e separados, cada um, por vírgula:
        1) Tipo propriamente dito:
            i) ***R*** para um acesso de leitura
            ii) ***W*** para um acesso de escrita
            iii) ***RW*** para um acesso de leitura/escrita
        2) Número da linha em que o respectivo comando *for* encontra-se no código, sendo um valor inteiro
        3) Número da coluna em que o respectivo comando *for* encontra-se no código, sendo um valor inteiro
        4) Nome do arquivo em que o acesso é realizado: ***'nome_arquivo'***

Para que o formato da saída seja melhor compreendido, visualizaremos um exemplo de um código escrito na linguagem C - Figura 2.1 - e a respectiva saída gerada pela ferramenta *CUIA* - Figura 2.2 após a realização da coleta de suas informações. O código-exemplo utilizado trata-se da aplicação de um algoritmo de ordenamento (*Bubble Sort*) a um array estático. É importante notar que, na saída, existem duas linhas referentes a variáveis de nome *x*. Isso se deve ao fato de existir, no código-fonte, duas variáveis com o mesmo nome, porém pertencentes a escopos diferentes.

```c
#include <time.h>

#define ARRAY_SIZE 20

int x;

void bubble(int *iarray)
{
    int x, y;

    for (x = 0; x < ARRAY_SIZE; x++)
        for (y = 0; y < ARRAY_SIZE - 1; y++)
            if(iarray[ y ] > iarray[ y + 1 ])
            {
                int holder;
                holder = iarray[ y + 1 ];
                iarray[ y + 1 ] = iarray[ y ];
                iarray[ y ] = holder;
            }
}

int main()
{
    int iarray[ARRAY_SIZE];

    // Cria semente
    srand((unsigned int)time(NULL));

    // Inicializa array
    for(x = 0; x < ARRAY_SIZE; x++)
        iarray[x] = (int)(rand() % 100);

    // Ordena o array com o bubble sort
    bubble(iarray);
}
```

Figura 2.1: Exemplo de Código-Fonte C Analisado

```
'holder' NOT_ARRAY 'int' BLOCK_SCOPE 14 13 'exemplo.c' -
'iarray' DYNAMIC_ARRAY 'int*' FUNC_SCOPE 'bubble' 'exemplo.c' (RW,12,9,'exemplo.c')
'x' NOT_ARRAY 'int' FUNC_SCOPE 'bubble' 'exemplo.c' -
'y' NOT_ARRAY 'int' FUNC_SCOPE 'bubble' 'exemplo.c' -
'iarray' STATIC_ARRAY 'int' 1 20 FUNC_SCOPE 'main' 'exemplo.c' (W,30,5,'exemplo.c')
'x' NOT_ARRAY 'int' GLOBAL_SCOPE 'exemplo.c' -
```

Figura 2.2: Exemplo de Saída da Ferramenta *CUIA*

3 *LEX* E *YACC*: AS FERRAMENTAS AUXILIARES UTILIZADAS

A leitura de um código-fonte para a extração de informações necessita de algum processo de reconhecimento e contextualização sintática de padrões. Para alcançar esse objetivo dentro do contexto da *CUIA*, foram utilizadas duas ferramentas comumente encontradas no desenvolvimento de compiladores e interpretadores: *Lex* e *Yacc*. Tais ferramentas, como serão explicadas adiante, são responsáveis pela construção de analisadores léxicos e sintáticos, sem que haja a necessidade de sua programação propriamente dita.

Primeiramente, neste capítulo, serão introduzidos alguns conceitos relativos aos processos de análise léxica e sintática. Posteriormente, serão citadas algumas ferramentas possíveis de serem utilizadas no contexto de tais análises e será explicado o funcionamento da escolha realizada. Por fim, será mostrada a sintaxe dos pseudocódigos usados nas listagens ao longo deste Trabalho.

3.1 A Análise Léxica e a Análise Sintática

O reconhecimento de padrões em códigos-fonte é realizado com a intenção de classificar conjuntos de caracteres com significados específicos em diferentes categorias, denominadas de *tokens*. Assim sendo, cada *token* representa um conjunto de cadeias de caracteres, permitindo que diferentes instâncias de um identificador possam ser mapeadas para o mesmo *token*, por exemplo. Esse conjunto de cadeias é descrito por uma regra chamada de *padrão* associado ao *token* de entrada, e tal regra é comumente definida por um formalismo denominado *expressão regular* (AHO; SETHI; ULLMAN, 1995).

Um analisador léxico possui a função de ler caracteres de uma entrada e agrupá-los em *tokens*. Basicamente, o algoritmo de reconhecimento de padrões realizado por tal analisador utiliza-se de autômatos finitos, que são mapeamentos diretos das *expressões regulares* que representam cada um dos *tokens* da linguagem em questão.

A estrutura sintática de um código-fonte, por sua vez, é representada por um formalismo conhecido como *gramática*. Este formalismo descreve naturalmente a estrutura hierárquica das construções de uma linguagem de programação, sendo as *gramáticas livre de contexto* as mais utilizadas para tal descrição. Essas gramáticas são compostas pelos quatro componentes listados a seguir (AHO; SETHI; ULLMAN, 1995):

1. Um conjunto de **tokens**, conhecidos como os símbolos terminais (tal como as palavras-chave *if* ou *for*).

2. Um conjunto de símbolos **não-terminais**.

3. Um conjunto de **produções**, onde uma produção consiste em um não-terminal, chamado de *lado esquerdo* da produção, uma seta e uma seqüência de tokens e/ou não-terminais, chamados de *lado direito* da produção.

4. Uma designação a um dos não-terminais como o **símbolo de partida**.

A listagem de todas as produções de uma gramática é feita de forma a especificar-se a sintaxe de uma linguagem. Por uma conveniência de notação, as produções com o mesmo não-terminal à esquerda podem ter todos os seus lados direitos agrupados, com os diferentes lados alternativos à direita separados pelo símbolo |. Ainda no contexto das gramáticas, uma *árvore sintática* mostra como o símbolo de partida de uma gramática deriva em uma cadeia de uma linguagem (AHO; SETHI; ULLMAN, 1995).

A análise sintática é o processo de se determinar se uma cadeia de tokens pode ser gerada por uma gramática. Basicamente, existem dois métodos para se implementar um analisador sintático: o *top-down* e o *bottom-up*. Em uma análise *top-down*, a construção da árvore inicia na raiz e prossegue em direção às folhas, enquanto que na *bottom-up* a construção se inicia nas folhas e prossegue em direção à raiz.

Independentemente do método utilizado para a construção de um analisador sintático, o conceito de *ação semântica* pode ser aplicado para adicionar trechos de código a uma gramática, de forma a produzir eventos em momentos apropriados da análise. Além disso, a utilização de *atributos* aos símbolos gramaticais pode fazer com que informações possam ser repassadas de um segmento de uma árvore sintática para outro. Essas duas possibilidades são exploradas no desenvolvimento da ferramenta criada neste Trabalho, de forma que a obtenção e o salvamento das informações das variáveis são incorporadas em ações semânticas na própria gramática. O uso de atributos, nesse caso, pode auxiliar nesse processo de obtenção de informações.

A construção de um analisador léxico e de um analisador sintático para uma linguagem é uma tarefa relativamente complexa e bastante suscetível a erros. Visando a automatização de tais processos, diversas ferramentas foram desenvolvidas ao longo dos últimos anos. A sessão a seguir descreverá algumas dessas ferramentas.

3.2 As Ferramentas Existentes

As ferramentas comumente encontradas para a geração de analisadores léxicos e sintáticos utilizam-se de uma especificação de entrada com regras que demonstram os tokens considerados (através de expressões regulares) e a gramática utilizada (com notações similares a BNF[1]). Além disso, para cada uma das regras de ambas as especificações, são permitidas a colocação de ações que são executadas quando o casamento dessas regras é realizado.

Uma das ferramentas existentes é o *ANTLR*[2], o qual é um gerador de *parser* que suporta a geração de código em linguagens como C, C++, Python, Java e C#. O analisador gerado por essa ferramenta utiliza-se de uma abordagem *top-down*, com um algoritmo LL(*) - uma extensão ao algoritmo LL(k) -, que permite um nível de lookahead arbitrário. Uma de suas vantagens é a existência de uma IDE, que possui um interpretador e um depurador embutidos.

[1] Backus-Naur Form
[2] *http://www.antlr.org/*

Uma outra opção é o *JavaCC*[3], o qual é uma ferramenta que gera analisadores léxicos e parsers *top-down* utilizando-se de um algoritmo LL(k). A linguagem em que os códigos são gerados por essa ferramenta é Java e algumas *features* podem ser alcançadas, tal como a geração de árvores de sintaxe e de documentação automática do código a partir dos arquivos de definição da gramática.

Por fim, duas ferramentas clássicas na área de compiladores são o *Lex* e o *Yacc*, os quais são geradores de analisadores léxicos e sintáticos, respectivamente. Essas ferramentas são comumente utilizas em conjunto, de forma que o procedimento para reconhecimento léxico, criado pelo *Lex*, é geralmente utilizado pelo procedimento de reconhecimento sintático, criado pelo *Yacc*. A abordagem utilizada, diferentemente das outras ferramentas, é *bottom-up* e o algoritmo utilizado é o LALR(1). Além disso, a linguagem comum em que os códigos são gerados por essas ferramentas é o C. Devido a familiaridade com tais ferramentas e a facilidade de se criar as especificações de entrada para as mesmas, escolheu-se ambas para auxílio no desenvolvimento da ferramenta *CUIA*.

3.3 As Ferramentas Escolhidas: *Lex* e *Yacc*

O *Lex* foi originalmente desenvolvido por Eric Schmidt e Mike Lesk na década de 70 nos Laboratórios Bell e é o gerador padrão de analisadores léxicos do ambiente Unix (MASON; BROWN, 1991). O funcionamento básico de tal ferramenta consiste na leitura de um arquivo contendo a especificação de expressões regulares para o reconhecimento de padrões e na geração de uma rotina escrita na linguagem C (denominada de **yylex**), que realiza o processo de análise léxica.

Além da classificação de cadeias de caracteres em tokens, a ferramenta *Lex* permite que, para cada token reconhecido na entrada, seja realizado um determinado conjunto de ações especificado na linguagem C, pelo usuário. Essa característica é extremamente importante, uma vez que ela permite que certos tratamentos possam ser realizados com a entrada, como o descarte de comentários e a diferenciação entre tokens representados por uma mesma expressão regular. Essa ferramenta, portanto, recebe como entrada uma especificação com um formato bem definido, a qual contém os mapeamentos de cada expressão regular com a ação a ser tomada logo após o reconhecimento de uma cadeia que satisfaça tal expressão.

O formato da especificação *Lex* é composto por três partes. A primeira parte é destinada para a seção de declarações de variáveis, constantes e definições regulares, as quais são componentes de expressões regulares que aparecem nas regras de tradução, colocadas na segunda parte. Essas regras são, na verdade, enunciados da forma

$$
\begin{gathered}
e_1 \ \{ação_1\} \\
e_2 \ \{ação_2\} \\
\ldots \\
e_3 \ \{ação_3\}
\end{gathered}
$$

onde cada e_i retrata uma expressão regular e cada $ação_i$ retrata a ação correspondente a ser tomada quando for realizado o reconhecimento do padrão representado por e_i. Por fim, a terceira parte é composta por funções escritas pelo usuário, em C, que podem ser chamadas dentro das ações existentes na seção das regras de tradução.

Basicamente, as expressões regulares utilizadas na especificação fornecida ao *Lex* fazem o reconhecimento de cada uma das palavras-chave contidas na linguagem C, além

[3] *https://javacc.dev.java.net/*

de identificadores e constantes numéricas, por exemplo. As ações básicas tomadas são o simples retorno do token identificado, portanto. Nesse caso, a parte léxica deste Trabalho será apenas mencionada e explicada quando as ações a serem realizadas para uma expressão regular exigirem um tratamento extra além do retorno do token reconhecido.

O *Yacc*, por sua vez, foi desenvolvido originalmente por Sthephen C. Johnson na AT&T para o sistema operacional Unix. O modo de sua utilização consiste no fornecimento de uma gramática como especificação de entrada e na geração de um procedimento (denominado de **yyparse**), escrito em C, que realize o processo de análise sintática. A especificação da gramática, porém, permite que o programador adicione ações semânticas a fim de realizar ações quando lhe for conveniente. Nesse caso, tais ações são realizadas simultaneamente ao processo de análise sintática. É importante destacar que esse fato foi explorado para o desenvolvimento da ferramenta *CUIA*, uma vez que a obtenção e o salvamento das informações das variáveis são realizados ao mesmo tempo em que a varredura e a estruturação sintática do código-fonte são realizadas.

A especificação fornecida ao *Yacc* é composta por três partes. A primeira delas é a seção de declarações, onde variáveis globais e que serão utilizadas ao longo de todo o *parser* são declaradas. É nesta seção onde as estruturas de dados utilizados pelo *CUIA* foram informadas. Além disso, as especificações de quais atributos existem em cada um dos símbolos terminais e não-terminais da gramática são colocadas nesta seção. A segunda seção, por outro lado, é a responsável pela definição da gramática propriamente dita e das ações semânticas. Por fim, a terceira e última seção é a responsável pela definição de rotinas de suporte às ações semânticas, de forma a modularizar a programação ao usuário.

O desenvolvimento da ferramenta *CUIA*, portanto, baseou-se na construção das especificações para o *Lex* e para o *Yacc*. Nesse caso, as ações semânticas incorporadas na especificação sintática são responsáveis pela obtenção das informações e manipulação das estruturas de dados para seu salvamento. Essas ferramentas auxiliares geram duas rotinas, em C, específicas para a análise léxica e sintática, denominadas de **yylex** e **yyparse**. O código gerado para tais analisadores e o código que manipula as estruturas de dados são então repassados a um compilador C (como o *cc*) para obter-se, ao final, a ferramenta desejada. O processo de desenvolvimento descrito encontra-se na Figura 3.1.

A gramática C utilizada neste Trabalho é a gramática C ANSI, publicada em 1985 por Jeff Lee e disponível online[4].

3.4 A Sintaxe dos Pseudocódigos

Basicamente, os pseudocódigos encontrados nas listagens deste Trabalho utilizam-se das seguintes regras:

- Os não-terminais da gramática são colocados em negrito.
- As ações semânticas são colocadas entre chaves ({ e }).
- Os terminais são tokens colocados em letras maiúsculas ou caracteres simples como o sinal de adição (+), por exemplo.
- As produções com o mesmo não-terminal à esquerda são separadas pelo símbolo |.
- O início das produções com um não-terminal à esquerda é dado pelo símbolo de dois pontos (:) .

[4] *http://www.lysator.liu.se/c/ANSI-C-grammar-y.html*

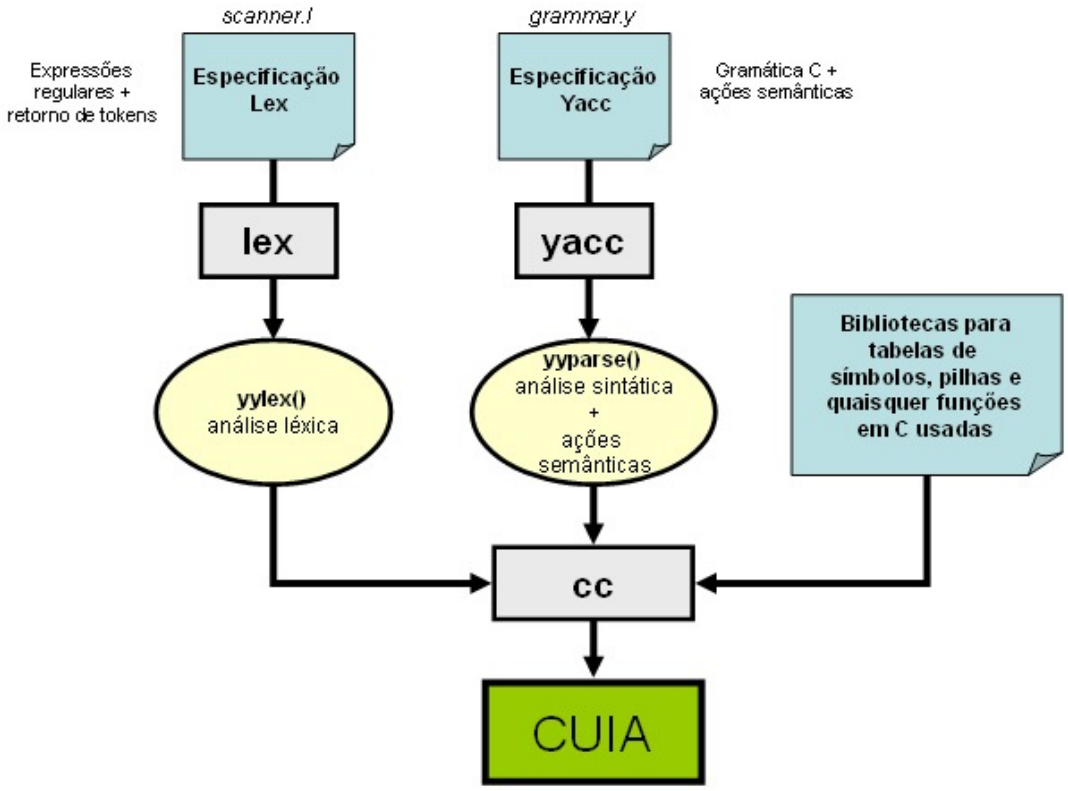

Figura 3.1: Fluxo de Construção da Ferramenta CUIA com o Auxílio das Ferramentas Lex e Yacc

- Para as produções que tiverem um não-terminal no lado direito com o mesmo nome do não-terminal do lado esquerdo, a maneira adotada para diferenciá-los quando forem referenciados nas ações semânticas é acoplar o símbolo *[E]* para o não-terminal à esquerda e o símbolo *[D]* para o não-terminal à direita.
- O acesso a um atributo de um não-terminal é feito utilizando-se o nome do não-terminal, seguido de um ponto (.), seguido do nome do atributo propriamente dito.

4 AS ESTRUTURAS DE DADOS UTILIZADAS

Para que as informações coletadas pelo processo de *parsing* fossem gradativamente salvos, tornou-se necessária a utilização de uma estrutura de dados prática e relativamente eficiente para tal função. Assim sendo, optou-se pela utilização de tabelas de símbolos implementadas como tabelas hash, uma vez que estas associam chaves de pesquisa a valores de forma a tornar a busca de informações rápida e são comumente utilizadas para guardar grandes volumes de informação. Essas vantagens fornecidas pela tabela hash foram decisivas para a escolha da estrutura de dados, uma vez que, a princípio, não se sabe o tamanho do código que será analisado pela *CUIA* nem a quantidade de variáveis que serão armazenadas em tais estruturas.

Cada entrada da tabela de símbolos é composta por campos que possuem o objetivo de guardar cada uma das informações mencionadas na seção 2.1. A Figura 4.1 relaciona cada um dos nomes dos campos de uma entrada com sua respectiva informação salva. Tais nomes serão referenciados no capítulo 5, onde a forma como as informações são obtidas será detalhadamente explicada.

É importante notar que o campo *escopo* trata-se, na verdade, de uma estrutura com três subcampos denominados de *nome*, *linha* e *coluna*. A necessidade de cada um desses subcampos será explicada adiante. Além disso, os campos *listaDimensoes* e *listaAcessos* são listas encadeadas de descritores de dimensões e de tipos de acesso, respectivamente.

Além de tabelas de símbolos e de listas encadeadas, diversas pilhas são utilizadas pela ferramenta como utensílio para o salvamento de informações. Para facilitar a compreensão das listagens e explicações que serão vistas no capítulo 5 , sobre a construção da ferramenta *CUIA*, criaram-se algumas funções que manipulam essas estruturas de dados e omitem detalhes de implementação. Dessa forma, a inserção de uma entrada em uma tabela de símbolos, por exemplo, será demonstrada pelo simples chamamento a uma função.

A seção a seguir listará cada uma das funções existentes no pseudocódigo das listagens deste Trabalho e explicará as suas respectivas funcionalidades.

4.1 As Funções Utilizadas

Neste Trabalho, quatro estruturas de dados básicas foram utilizadas: tabelas de símbolos, pilhas, listas e filas. A manipulação dessas estruturas foi incorporada a um conjunto de funções, as quais serão detalhadas nas subseções a seguir.

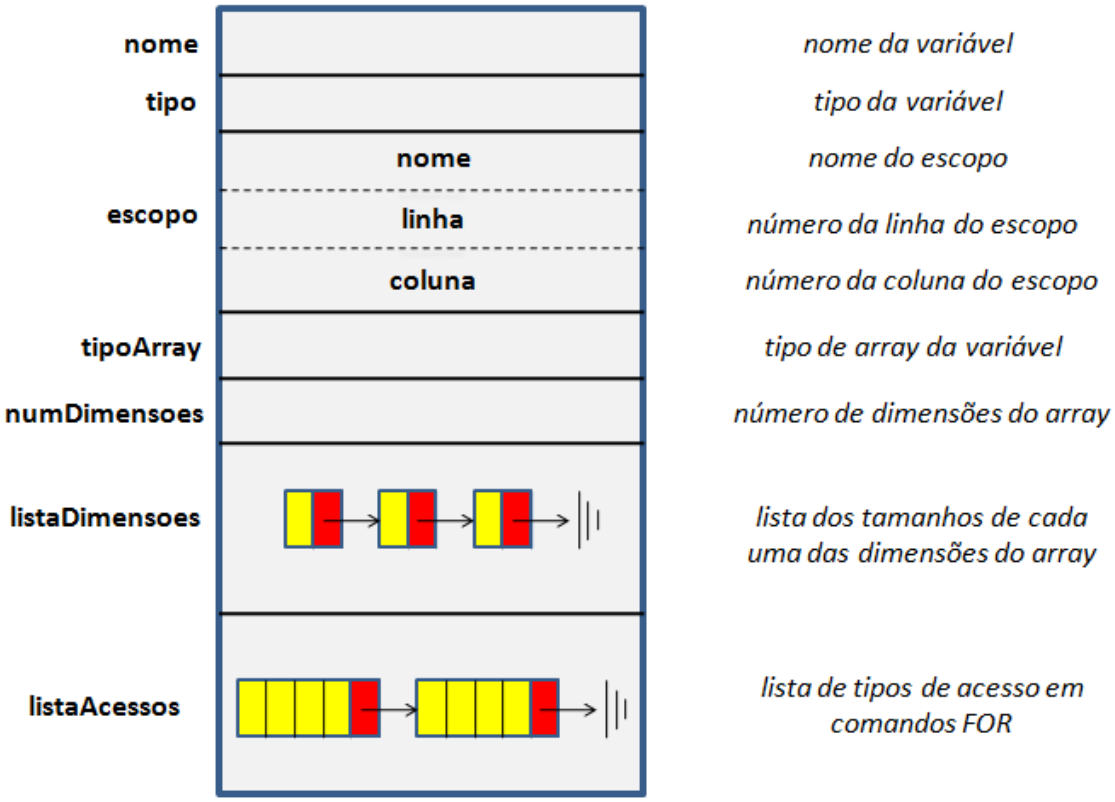

Figura 4.1: Entrada de Uma Variável e Seus Respectivos Nomes de Campos Utilizados

4.1.1 cria_entrada ( )

Função que cria a estrutura de dados relativa a uma entrada de uma tabela de símbolos e retorna tal entrada.

4.1.2 insere_entrada (tabela *t*, entrada *e*)

Função que insere uma entrada *e* em uma tabela de símbolos *t*.

4.1.3 retira_entrada (tabela *t*, entrada *e*)

Função que retira uma entrada *e* de uma tabela de símbolos *t*.

4.1.4 imprime_tabela (tabela *t*)

Função que imprime todas as entradas existentes de uma tabela *t* na saída padrão da ferramenta. A impressão dos dados é feita no formato de saída da *CUIA*, concatenando-se os dados ao final daqueles já impressos, eventualmente, por outra chamada anterior a essa função.

4.1.5 empilha (pilha *p* , descritor *d*)

Função que empilha um descritor *d* em uma pilha *p*. As pilhas utilizadas neste Trabalho empilharão descritores de escopo e de arquivo.

4.1.6 desempilha (pilha *p*)

Função que desempilha um descritor de uma pilha *p*, retirando-se o elemento do topo da pilha.

4.1.7 topo_pilha (pilha *p*)

Função que retorna o descritor existente no topo da pilha *p*. É importante notar que o elemento do topo não é retirado da pilha.

4.1.8 pilha_vazia (pilha *p*)

Função que retorna um valor *booleano*, informando se a pilha *p* encontra-se vazia.

4.1.9 coloca_fila (fila *f* , string *s*)

Função que insere um *string s* em uma fila *f* de *strings*.

4.1.10 concatena_elementos_fila (fila *f*)

Função que concatena cada um dos *strings* presentes em uma fila *f*, intercalando-os com um caractere *espaço*. O retorno da função é o *string* final formado pela concatenação.

4.1.11 concatena_asteriscos (int *n* , string *s*)

Função que concatena *n* caracteres asterisco (*) com o *string s*. O retorno da função é o *string* final formado pela concatenação. Essa função será útil para o tratamento do tipo de uma variável. Exemplo de uso: **concatena_asteriscos** (*2* , *"int"*) retorna o *string* "**int".

4.1.12 cria_elemento_lista_dimensoes (int *n*)

Função que cria um elemento de uma lista de inteiros. O valor inteiro do elemento criado é *n*. O elemento, no momento da chamada a essa função, é apenas criado e retornado, sem a inserção em uma lista específica. O nome *lista_dimensoes* da função se deve ao fato de elementos desse tipo serem usados para criar a lista de tamanho das dimensões de um array.

4.1.13 cria_elemento_lista_acessos (int *n* , int *linha* , int *coluna* , string *arquivo*)

Função que cria um elemento de uma lista de tipos de acesso. O valor *n* trata-se do tipo do acesso, enquanto que *linha* e *coluna* são valores inteiros que especificam o número da linha e da coluna do escopo em que o acesso é realizado, respectivamente. O string *arquivo* é o nome do arquivo em que o acesso é realizado. O elemento, no momento da chamada a essa função, é apenas criado e retornado, sem a inserção em uma lista específica. O nome *lista_acessos* da função se deve ao fato de elementos desse tipo serem usados para criar a lista de tipos de acesso de um array em comandos *for*.

4.1.14 insere_lista (lista *l* , elemento *e*)

Função que insere um elemento *e*, criado ou pela função *cria_elemento_lista_dimensoes* ou pela *cria_elementos_lista_acessos*, em uma lista *l*.

4.1.15 existe_elemento_lista (lista *l* , int *linha* , int *coluna* , string *arquivo*)

Função que verifica se já existe um elemento na lista *l* com o número de linha, coluna e nome do arquivo iguais a *linha*, *coluna* e *arquivo*, respectivamente. O retorno é um valor *booleano* indicando tal existência ou não.

5 A CONSTRUÇÃO DA FERRAMENTA CUIA

CUIA é a ferramenta desenvolvida ao longo deste Trabalho de Conclusão que tem como finalidade a automatização do processo de obtenção de informações relativas a variáveis presentes em códigos desenvolvidos na linguagem C.

Neste capítulo serão descritos quais os métodos utilizados para a construção da ferramenta *CUIA*. Como mencionado anteriormente no capítulo 3, foram utilizadas, para este Trabalho, duas ferramentas clássicas na área de compiladores, *Lex* e *Yacc*, as quais são responsáveis pela construção de um analisador léxico e sintático, respectivamente. A maneira como as informações são coletadas é baseada, principalmente, no acréscimo de ações semânticas na gramática utilizada como especificação de entrada para a ferramenta *Yacc*, além de tratamentos léxicos feitos na especificação de entrada para o *Lex*. Nesse contexto, a maior parte das explicações contidas neste capítulo são baseadas na construção de tais ações semânticas e tratamentos léxicos.

Primeiramente, serão detalhadas as formas como a ferramenta obtém cada uma das informações listadas no capítulo 2. Por fim, serão observados os tratamentos técnicos feitos para diretivas de pré-processamento, como a diretiva *#define*.

5.1 A Obtenção das Informações

A linguagem de programação C - padrão ANSI, fornece a possibilidade de o programador declarar as variáveis que desejar no início da seção referente ao escopo em que se está trabalhando, a fim de que todas as variáveis sejam previamente conhecidas antes da análise, pelo compilador, de qualquer instrução que possa fazer menção às mesmas (HARBISON; STEELE, 2002). Essa característica é refletida na gramática utilizada para a *CUIA*, uma vez de que ela demonstra, através de suas produções específicas para o englobamento da seção de declaração de variáveis, tais restrições fornecidas pelo padrão ANSI.

Praticamente todas as informações mencionadas na seção 2.1 são obtidas na própria declaração das variáveis, com a excessão da lista com os tipos de acesso que são feitos a arrays em comandos do tipo *for*. Isto significa dizer que, dentro de um escopo específico (como uma função) e dentro de um arquivo específico, quando o processo de análise sintática gerado para o *CUIA* encontrar o nome de uma variável na seção de declarações daquele escopo, pode-se obter, naquele momento, praticamente todas as informações necessárias para a variável em questão. Para a obtenção de tais informações, foram incluídas ações semânticas em produções específicas para a declaração de variáveis, com o objetivo de coletar essas informações apropriadamente e salvá-las na tabela de símbolos.

No caso da lista com os tipos de acesso, a inclusão de ações semânticas para tal tratamento foi realizada fora da seção de declarações, uma vez que os acessos de escrita e

leitura de uma variável são realizados em segmentos de código localmente separados de sua declaração, os quais utilizam-se de comandos específicos para o manuseio de variáveis na memória. Essa divisão de tipos de informação obtidas de acordo com as produções da gramática C podem ser vistas na Figura 5.1.

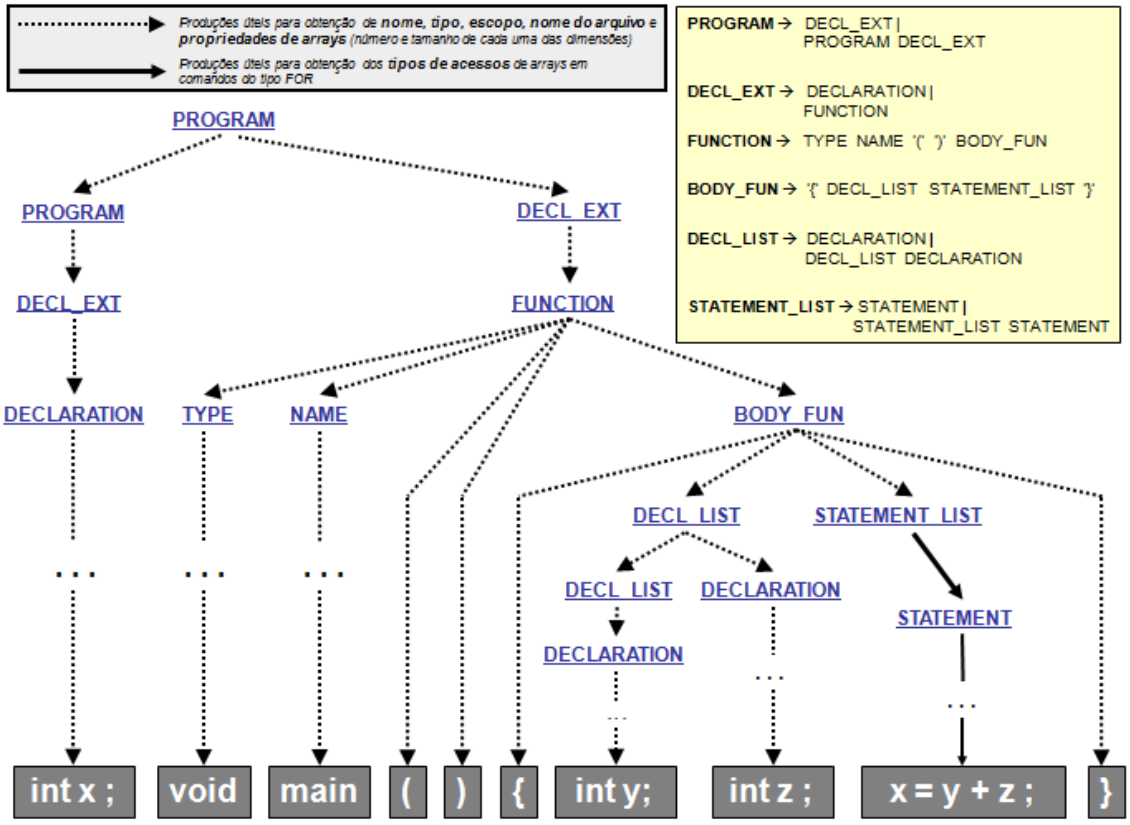

Figura 5.1: Recuperação de Informações Dividida em Tipos de Produções

A gramática exposta na Figura 5.1 é um subconjunto da gramática C original utilizada para este Trabalho. Assim sendo, foram citadas as principais produções que separam sintaticamente um código C em blocos funcionalmente bem definidos. Basicamente, um programa escrito em C é composto por uma lista de declarações externas, ou globais. Cada uma dessas declarações externas pode ser uma declaração propriamente dita ou uma definição de uma função. A declaração em si envolve produções que tratam a estrutura sintática de cada um dos tokens relativos a ela, como o tipo e o nome dos identificadores declarados. Essas produções não foram citadas na Figura 5.1, porém serão analisadas nas subseções seguintes, quando a recuperação das informações for explicada detalhadamente. O outro tipo de declaração externa, a definição de uma função, é composta por um tipo de retorno, um nome e um corpo de função. O corpo, por conseguinte, é formado por uma lista de declarações e uma lista de comandos, os quais são formados por produções que envolvem principalmente expressões aritméticas, instruções de iteração, de seleção e de controle de fluxo. Tais produções são responsáveis pela inferência das informações relativas aos tipos de acessos a arrays e encontram-se localizadas separadamente, na gramática, das demais produções responsáveis pela obtenção de informações como o escopo ou o nome de uma variável. O código-exemplo utilizado para demonstrar a diferenciação das derivações da gramática é o apresentado na listagem 5.1.

```
1 int x;
```

```
2
3  void main()
4  {
5          int y;
6          int z;
7          x = y + z;
8  }
```

Listagem 5.1: Exemplo de Código C

5.1.1 Obtenção do Nome da Variável

O nome de uma variável é a informação mais simples de ser recolhida, uma vez que ela é, na verdade, obtida no analisador léxico e repassada ao analisador sintático como o valor do atributo *nome* do token *IDENTIFIER* retornado. Assim sendo, na especificação léxica, no conjunto de ações referentes à expressão regular correspondente a identificadores, devem ser repassados para o analisador sintático o token correspondente a um identificador e a cadeia de caracteres que foi reconhecida, a qual nada mais é que o nome da variável em questão. O analisador sintático passa, dessa forma, a ter a possibilidade de acessar o nome de um identificador em quaisquer reduções de produções que contenham algum identificador em seus símbolos à direita.

Basicamente, a produção mais importante da gramática para a declaração de uma variável é a listada abaixo:

$$direct\_declarator \longrightarrow IDENTIFIER$$

A partir dessa produção, a ferramenta passa a ter o conhecimento da existência da declaração de um identificador no código-fonte. Além disso, tal produção é apenas reduzida quando o processo de análise sintática estiver varrendo a seção de declarações pertencente a um escopo específico (como uma função, por exemplo). Para cada variável encontrada na varredura dessa seção de declarações, a produção mencionada é reduzida e a ação semântica correspondente é executada. Esta ação semântica, como pode ser visto no pseudocódigo da listagem 5.3, deve realizar a criação de uma nova entrada na tabela de símbolos relativa ao escopo atual do processo de análise e atualizar seus campos com as informações já obtidas até o momento. O nome do identificador é uma das informações que podem ser apropriadamente colocadas na entrada, uma vez que o mesmo pode ser acessado pelo valor do campo *nome* do token *IDENTIFIER*. Esse valor foi repassado da análise léxica, que realizou o casamento do padrão relativo a um identificador e colocou o lexema desse padrão no campo *nome* do token em questão (listagem 5.2).

```
1  {letra}({letra}|{digito})*
2
3    {
4      token = IDENTIFIER;
5      token.nome = lexema_reconhecido;
6      return ( token );
7    }
```

Listagem 5.2: Especificação da Ação da Análise Léxica para Casamentos de Identificadores

Nota-se que, na listagem 5.2, a expressão regular indica que o reconhecimento de tokens do tipo identificador é realizado quando encontra-se uma letra seguida de zero

ou mais letras ou digitos. A partir do momento que tal padrão é casado, indica-se que o token em questão é um *IDENTIFIER* e o valor de seu campo *nome* é a sequencia de caracteres reconhecida. Já na análise sintática, especificamente na listagem 5.3, percebe-se que a ação semântica relativa a descoberta da declaração de um identificador deve criar uma entrada para uma tabela de símbolos e atualizar a informação do nome com o valor do token *IDENTIFIER* obtido. A inserção da entrada é realizada na tabela de símbolos relativa ao escopo atual do código. Como podem ocorrer escopos aninhados, utiliza-se uma pilha de escopos denominada de *pilha_escopos*. Tal pilha será explicada na subseção 5.1.3, mas, por ora, basta saber e presumir que no topo da pilha sempre existirá um descritor do escopo atual do código, com um campo denominado *tabela* que apontará para uma tabela de símbolos própria daquele escopo. Por fim, é importante notar que, na linha 12 dessa mesma listagem, a entrada da tabela é repassada para o não-terminal superior da árvore de derivação na forma do atributo sintetizado *entrada*. Tal ação é executada para que, no decorrer da análise *bottom-up* do código, seja possível reacessar a entrada inserida na tabela, a fim de que novas informações possam ser atualizadas na mesma.

```
1  direct_declarator : IDENTIFIER
2
3    {
4        entrada = criaEntrada();
5        entrada.nome = IDENTIFIER.nome;
6
7        /** Insere outras informacoes na entrada **/
8
9        escopo_atual = topo_pilha( pilha_escopos );
10       insere_entrada ( escopo_atual.tabela , entrada );
11
12       direct_declarator.entrada = entrada ;
13   }
```

Listagem 5.3: Ação Semântica para Criação de Nova Entrada na Tabela de Símbolos e Obtenção do Nome da Variável

5.1.2 Obtenção do Tipo da Variável

A informação do tipo de uma variável permite a um compilador inferir tamanhos de armazenamento de dados e realizar tratamentos semânticos, a fim de detectar erros que, eventualmente, o processo de análise sintática possa não ter capturado, já que tal análise limita-se apenas a realização da verificação estrutural de um código.

Na linguagem de programação C, o tipo de uma variável pode ser constituído de um ou mais tokens, como nos exemplos de declarações da listagem 5.4. No caso da variável *x* dessa listagem, o seu respectivo tipo é constituído de dois tokens distintos: *unsigned* e *int*. Além disso, o mesmo conjunto de tokens que define um tipo pode estar relacionado à declaração de mais de um identificador, como pode ser visto na linha 1 da mesma listagem. Naquele caso, os identificadores *a*, *b* e *c* possuem o mesmo tipo: *long long int*.

```
1 long long int a,b,c;
2 unsigned int x;
3 static float y;
4 char z;
```

Listagem 5.4: Exemplos de Declarações de Variáveis na Linguagem C

O fato de que a informação do tipo de uma variável está sempre presente antes dos respectivos nomes de identificadores relacionados exige um tratamento diferente desta informação pela ferramenta *CUIA*. Dessa forma, é necessário que, no momento que a análise sintática realizar a varredura do tipo em uma declaração, sejem salvos em uma estrutura de dados cada um dos tokens que representam esse tipo. Esse procedimento é feito para que, quando os nomes de identificadores forem posteriormente varridos, possa ser feita a correta atualização do tipo nas respectivas entradas criadas na tabela de símbolos.

A estrutura de dados utilizada para guardar o conjunto de tokens relativos a um tipo trata-se de uma fila, denominada de *fila_tipos*. Nesse caso, após cada redução de produções que derivam em um terminal específico para a declaração de tipos, coloca-se no fim da fila o *string* que representa tal terminal. Tais ações semânticas são exemplificadas na listagem 5.5. As ações semânticas para os demais tokens responsáveis pela tipagem de uma variável seguem o mesmo padrão, sendo desnecessário listá-las exaustivamente aqui.

```
1  type_specifier
2    : VOID     { coloca_fila ( fila_tipos , "void" );  }
3    | CHAR     { coloca_fila ( fila_tipos , "char" );  }
4    | SHORT    { coloca_fila ( fila_tipos , "short" ); }
5    | INT      { coloca_fila ( fila_tipos , "int" );   }
6    | LONG     { coloca_fila ( fila_tipos , "long" );  }
7    | FLOAT    { coloca_fila ( fila_tipos , "float" ); }
8          ...
```

Listagem 5.5: Ações Semânticas para o Salvamento dos Tokens do Tipo de uma Variável

Partindo desse princípio, no momento em que o identificador relativo ao nome de uma variável for varrido e sua respectiva entrada na tabela de símbolos criada, poderá ser atualizada a informação do tipo verificando-se o conteúdo de *fila_tipos*. Nota-se que o conteúdo da fila não é retirado, uma vez que seu valor deve ser mantido para que todos os identificadores presentes na lista de declaradores sejem corretamente atualizados com a informação do tipo da variável. A fila só será esvaziada, portanto, quando a declaração for completamente varrida, a fim de que ela possa ser reaproveitada para a próxima declaração encontrada no código.

A inclusão da ação para atualizar a entrada da tabela de símbolos com o tipo da variável é descrita na linha 6 da listagem 5.6. Nota-se que a função *concatena_elementos_fila* é responsável por concatenar cada um dos *strings* presentes do início da fila até o seu final (sem apagar o seu conteúdo), retornando um *string* único com a informação do tipo básico da variável. Outro ponto importante a ser considerado é que, no momento de reconhecimento do identificador, não se sabe quantos níveis de ponteiros (número de asteriscos) existem acoplados ao tipo básico da variável. Tal reconhecimento será detalhado na subseção 5.1.5, onde o *tipo de array* e o *número de níveis de ponteiros* serão obtidos e devidamente atualizados na respectiva entrada da tabela.

```
1  direct_declarator : IDENTIFIER
2
3    {
4        entrada = criaEntrada();
5        entrada.nome = IDENTIFIER.nome;
6        entrada.tipo = concatena_elementos_fila ( fila_tipos );
7
8        /** Insere outras informacoes na entrada **/
9
10       escopo_atual = topo_pilha( pilha_escopos );
11       insere_entrada ( escopo_atual.tabela , entrada );
```

```
12
13        direct_declarator.entrada = entrada ;
14    }
```

Listagem 5.6: Ação Semântica para Atualização do Tipo de uma Variável

5.1.3 Obtenção do Escopo da Variável

Todas as variáveis em um código-fonte C possuem uma determinada área do código em que as mesmas são *visíveis*, permitindo duas características importantes: a distinção de permissões de acesso a variáveis de acordo com o local de sua utilização e a existência de variáveis com o mesmo nome no código, porém com visibilidades diferentes. Essa visibilidade trata-se do escopo, ou seja, a área do código em que é permitido o acesso a uma variável específica.

No caso da ferramenta *CUIA*, definiu-se três possibilidades de escopo, as quais encontram-se listadas a seguir:

- Escopo *global*, em que uma variável possui visibilidade em todo o código do arquivo, a partir do local de sua declaração
- Escopo local a uma *função*, em que uma variável possui visibilidade em toda uma função, a partir do local de sua declaração
- Escopo local a um *bloco*, em que uma variável possui visibilidade em todo um bloco, a partir do local de sua declaração

De acordo com o tipo de escopo apropriado a uma variável, diferentes saídas são fornecidas pela ferramenta para este tipo de informação. A relação da saída para cada tipo de escopo é apresentada na tabela abaixo. É importante notar que para o escopo global, não é necessária nenhuma outra informação para inferir a visibilidade da variável, com excessão do nome do arquivo em que ela foi declarada - informação esta adquirida e listada posteriormente. Para o escopo local a uma função, informa-se apenas o nome da função em que a variável é visível, enquanto que no caso de um escopo de bloco, são listadas as coordenadas - linha e coluna no código - do início do bloco.

Tipo de Escopo	**Saída**
Global	*GLOBAL_SCOPE*
Função	*FUNC_SCOPE* <nome_funcao>
Bloco	*BLOCK_SCOPE* <linha_inicio> <coluna_inicio>

Basicamente, um código C é composto por declarações e definições de funções. Nesse caso, o escopo de qualquer função está incluso no escopo global, uma vez que a área de código relativa a uma função está inclusa na área visível pelo escopo global. Isso significa dizer que uma variável global, declarada fora do escopo de uma função, pode ser acessada dentro da função, pois sua visibilidade inclui os escopos menores. Da mesma forma, uma variável declarada localmente a uma função pode ser acessada dentro dos sub-blocos existentes dentro dessa função, e assim por diante. Essa relação existente entres escopos nos permite inferir que o escopo global é o mais abrangente, seguido de escopos de funções e , posteriormente, escopos de blocos e sub-blocos existentes.

Para que a informação do escopo de uma variável fosse corretamente coletada na sua declaração, era necessário que a ferramenta tivesse o conhecimento, em cada momento da

varredura, do nível mais próximo de escopo em que a análise se encontrava. Para que isso fosse possível, e baseado na relação de inclusão existente entre os escopos, foi utilizada uma pilha denominada de *pilha_escopos* que continha, em cada elemento, um descritor dos escopos inclusos até o momento pela ferramenta. O gerenciamento dessa pilha pela ferramenta obedece as seguintes regras:

- Antes do início da análise, inclui-se no topo de *pilha_escopos* um descritor do escopo global.
- Quando a varredura passar a identificar o início de uma definição de uma função, inclui-se no topo de *pilha_escopos* um descritor do escopo local a função em questão.
- Quando a varredura passar a identificar o início de um bloco, dentro de uma definição de função, inclui-se no topo de *pilha_escopos* um descritor do escopo do bloco em questão.
- Quando a varredura passar a identificar o início de um sub-bloco, dentro de um bloco já existente, inclui-se no topo de *pilha_escopos* um descritor do novo sub-bloco em questão (e assim sucessivamente).
- Para cada fim de sub-bloco, bloco ou função, desempilha-se um elemento do topo de *pilha_escopos*.

Seguindo as regras estipuladas acima, é garantido que, no momento da varredura de uma declaração de uma variável no código, a informação de seu escopo estará presente no topo de *pilha_escopos*, não necessitando de nenhum outro tratamento para a obtenção de tal informação. Para que esse processo funcione, o descritor de escopo precisa conter quatro informações: o nome do escopo (no caso de ser uma função), o número da linha e coluna do escopo (no caso de ser um bloco) e um ponteiro para uma tabela de símbolos. Nesse caso, cada escopo possui sua própria tabela de símbolos, de forma que a inclusão da entrada relativa a uma variável sempre ocorrerá na tabela de símbolos que encontra-se no descritor do topo de *pilha_escopos*. No momento em que o escopo passar a não existir mais na varredura, como o término da análise de uma função, por exemplo, deve-se desempilhar um descritor de escopo da pilha e listar na saída todas as informações das entradas da tabela de símbolos relativa a esse escopo[1]. A atualização da ação semântica da listagem 5.3 com a correta inclusão das informações de escopo na entrada da tabela de símbolos pode ser vista na listagem 5.7.

```
1  direct_declarator : IDENTIFIER
2
3    {
4        entrada = criaEntrada();
5        entrada.nome = IDENTIFIER.nome;
6        entrada.tipo = concatena_elementos_fila ( fila_tipos );
7
8        /** Insere outras informacoes na entrada **/
9
10       escopo_atual = topo_pilha ( pilha_escopos );
```

[1]Como a finalização da varredura do escopo *global* é apenas realizada no término da análise do código, todas as informações relativas a variáveis globais sempre se encontrarão no final da listagem emitida na saída da ferramenta.

```
11        entrada.escopo.nome = escopo_atual.nome;
12        entrada.escopo.linha = escopo_atual.linha;
13        entrada.escopo.coluna = escopo_atual.coluna;
14
15        insere_entrada ( escopo_atual.tabela , entrada );
16
17        direct_declarator.entrada = entrada ;
18    }
```

Listagem 5.7: Inserção do Escopo na Entrada da Tabela de Símbolos

5.1.3.1 O Gerenciamento de pilha_escopos

A principal tarefa enfrentada para a obtenção do escopo de uma variável reside, portanto, no gerenciamento de *pilha_escopos*. Para que o processo funcione, tal gerenciamento deve seguir as regras mencionadas anteriormente. Assim sendo, devem ser incluídas ações semânticas que realizem a inserção e a retirada de descritores de escopos da pilha, de forma a manter a coerência do estado da varredura com o respectivo escopo em que ela se encontra.

Num primeiro momento, deve-se empilhar um descritor do escopo global na base da pilha, antes do início do processo de análise do código. Após feito isso, é necessário que se garanta que a cada varredura de uma nova definição de função e de um novo bloco, sejam empilhados um novo descritor na variável *pilha_escopos*.

No primeiro caso, deve-se observar que o nome da função e a sua respectiva lista de parâmetros formam um segmento de código derivado a partir do não-terminal *declarator*, que nada mais é que o declarador existente antes do corpo da função. Esse segmento de código pode ser derivado de três formas diferentes, de forma que a inserção de um novo descritor de escopo na pilha deve ser realizada em três produções distintas, como demonstrado na listagem 5.8. Nessa listagem, o não-terminal *direct_declarator* deverá, obrigatoriamente, ter derivado em um identificador que representa o nome da função. Dessa forma, permite-se que se possa acessar o nome da função através do atributo sintetizado *entrada* desse não-terminal, o qual foi preenchido adequadamente na ação semântica da listagem 5.3. É importante notar que, da maneira como o processo foi formulado até agora, o nome da função também foi inserido em uma tabela de símbolos anteriormente, quando a produção referente ao reconhecimento da declaração de um identificador foi reduzida. Essa inserção ocorre pois, no momento daquela redução, não se tem o conhecimento de que o identificador trata-se do nome de uma variável ou de uma função. Deve-se, portanto, eliminar tal entrada, como também pode ser visto na listagem 5.8.

```
1  direct_declarator
2    : direct_declarator ( )
3
4      {
5        escopo = cria_descritor_escopo( direct_declarator[D].entrada.nome
             , INDEFINIDO , INDEFINIDO );
6        empilha ( pilha_escopos , escopo );
7        retira_entrada( escopo.tabela , direct_declarator[D].entrada );
8      }
9
10   | direct_declarator ( identifier_list )
11
12     {
13       escopo = cria_descritor_escopo( direct_declarator[D].entrada.nome
             , INDEFINIDO , INDEFINIDO );
```

```
14        empilha ( pilha_escopos , escopo );
15        retira_entrada( escopo.tabela , direct_declarator[D].entrada );
16      }
17
18    | direct_declarator (
19
20      {
21        escopo = cria_descritor_escopo( direct_declarator[D].entrada.nome
              , INDEFINIDO , INDEFINIDO );
22        empilha ( pilha_escopos , escopo );
23        retira_entrada( escopo.tabela , direct_declarator[D].entrada );
24      }
25
26        parameter_type_list )
```

Listagem 5.8: Ações Semânticas para Empilhar o Escopo de uma Função

É importante notar que, conforme pode ser visto na listagem 5.8, empilha-se um descritor de escopo no instante da análise em que o reconhecimento do nome da função já foi realizado. Porém, nesse mesmo intante, ainda não ocorreu a varredura de nenhuma declaração de variável presente tanto na possível lista de parâmetros como no corpo da função. Esse é o motivo para que a ação semântica da terceira produção esteja presente antes do não-terminal *parameter_type_list*, uma vez que ele é o responsável pela derivação dos parâmetros da função, que são variáveis de escopo local a função sendo analisada. Além disso, o descritor do escopo é preenchido com números de linha e coluna indefinidos, já que essas informações não são necessárias para esse tipo de escopo. A função *cria_descritor_escopo*, que não foi listada no capítulo 3 por razões didáticas, possui a função de criar o descritor do novo escopo e a sua respectiva tabela de símbolos.

O escopo empilhado no início da varredura de uma definição de uma função deve ser desempilhado quando a análise da função for finalizada. Além disso, deve-se emitir na saída a listagem das informações contidas na tabela de símbolos desse escopo, uma vez que todas as variáveis relativas a tal escopo já foram varridas e suas informações devidamente preenchidas na tabela. Essas ações podem ser colocadas na produção que deriva em um *function_definition*, garantindo-se, assim, que elas sejam executadas logo após a finalização da varredura de uma definição de função (listagem 5.9).

```
1  external_declaration
2    : function_definition
3
4      { escopo = desempilha ( pilha_escopos );
5        imprime_tabela ( escopo.tabela );      }
```

Listagem 5.9: Ações Semânticas para Desempilhar o Escopo de uma Função

No caso de um escopo de um bloco, deve-se empilhar um descritor após o reconhecimento do caractere *{*, o qual é o delimitador de início de bloco em C. Basicamente, todo bloco é originado a partir do não-terminal *compound_statement*, o qual é o responsável por derivar todo o conjunto de declarações e comandos existentes dentro de um bloco específico. Dessa forma, deve-se acrescentar ações semânticas que realizem esse empilhamento nas produções que derivam desse não-terminal, como pode ser visto na listagem 5.10. Nota-se que o teste *nao_eh_inicio_de_bloco_de_funcao* é realizado para que não sejam criados dois descritores de escopo de função na pilha, uma vez que tal escopo já foi anteriormente criado quando o declarador da função foi reconhecido. Outro ponto importante é a inclusão de um não-terminal *X* na gramática, o qual deriva na con-

tinuação de todos os possíveis blocos cujo descritor de escopo necessita ser criado. Essa inclusão do não-terminal foi feita para se retirar conflitos no processo de análise sintática pelo *Yacc*, não se alterando a linguagem aceita pela gramática original. Ainda sobre esta listagem, deve-se mencionar que os campos *linha* e *coluna* do token de abertura de bloco são preenchidos no analisador léxico e repassados para o analisador sintático.

```
1  compound_statement
2    : {
3
4      { if ( nao_eh_inicio_de_bloco_de_funcao )
5        {
6          escopo = cria_descritor( INDEFINIDO , '{'.linha , '}'.coluna );
7          empilha( pilha_escopos , escopo );
8        }
9      }
10
11     X
12   ;
13
14 X
15   : }
16   | declaration_list }
17   | declaration_list statement_list }
18   | statement_list }
19   ;
```

Listagem 5.10: Ações Semânticas para Empilhar o Escopo de um Bloco

Por fim, deve-se desempilhar o descritor do escopo ao ser reconhecido o caractere *}*, que é o delimitador de fim de bloco em C, e emitir na saída as informações da tabela de símbolos relativa a esse escopo. Para isso, basta adicionar uma ação semântica após a derivação do não-terminal *statament* em um *compund_statement*, uma vez que tal redução será apenas realizada após a finalização da varredura de um bloco. Tal ação pode ser vista na listagem 5.11.

```
1  statement
2    : compound_statement
3
4      { escopo = desempilha ( pilha_escopos );
5        imprime_tabela ( escopo.tabela );      }
```

Listagem 5.11: Ação Semântica para Desempilhar o Escopo de um Bloco

O gerenciamento de *pilha_escopos* pode ser melhor observada na Figura 5.2, onde a situação da pilha é colocada em cada instante da varredura de um exemplo de código C. Para cada token lido do código, indica-se qual seria o formato da pilha naquele dado instante da análise, de forma a ratificar a idéia de que o escopo presente no topo da pilha será sempre o escopo correto das variáveis sendo lidas no código. Nesse caso, no exemplo mostrado, as variáveis *x* e *w* possuirão escopo *global*, enquanto que as variáveis *y* e *z* serão de escopo local a função *foo*.

5.1.4 Obtenção do Nome do Arquivo em que a Variável é Declarada

Um código completo, em C, não necessariamente encontra-se escrito em apenas um arquivo. Na verdade, em aplicações reais, é comum que diferentes segmentos de código com diferentes funções sejem separados em diferentes arquivos, a fim de modularizar o

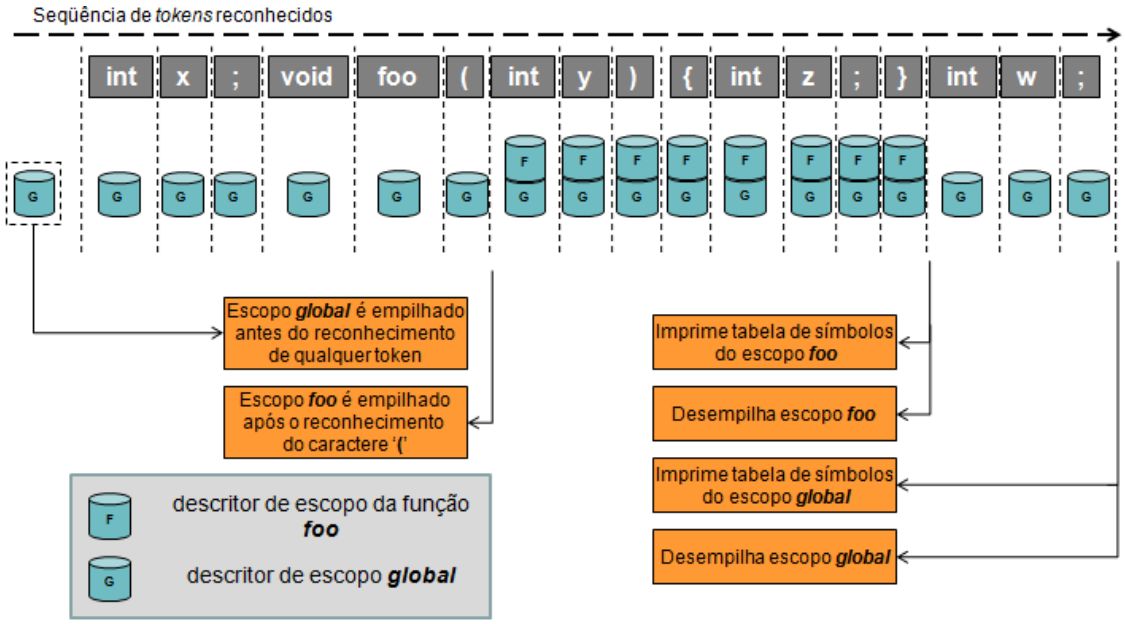

Figura 5.2: Seqüência de Varredura de Código C com Respectiva Situação da Pilha de Escopos

programa e de facilitar a sua manutenção e entendimento. Tipicamente, a indicação de que o código encontra-se dividido em vários arquivos é informada a um pré-processador C através da diretiva *#include*. Da mesma forma que um primeiro arquivo pode incluir um segundo através desta diretiva, este segundo pode incluir um teceiro, de tal forma que não existe, teoricamente, um limite para a quantidade de arquivos possíveis de serem incluídos em um programa C.

No contexto da ferramenta *CUIA*, tornou-se necessário o tratamento de tal questão. O usuário final, portanto, fornece como entrada para a ferramenta apenas os arquivos principais do código que deseja obter informações. Todos os demais arquivos incluídos a partir destes arquivos e presentes no mesmo diretório em que a ferramenta se encontra (ou seja, apenas os arquivos incluídos pela diretiva *#include* que estejam entre *aspas*) são também percorridos e analisados. Nesse caso, os arquivos de bibliotecas como *stdio.h* e *time.h* acabam não sendo varridos, pois comumente não são do interesse da aplicação ou usuário externo da ferramenta, cuja função é a análise das variáveis presentes no código propriamente dito do programa.

O fluxo de execução da ferramenta utiliza-se de uma pilha chamada *pilha_arquivos*, a qual possui a função de guardar os nomes dos arquivos sendo analisados pela ferramenta. Em um primeiro momento, no início da execução do *CUIA*, a pilha contém N elementos, onde N é a quantidade de arquivos passados como parâmetro pelo usuário para a ferramenta. No instante em que o analisador léxico encontra um segmento de código relativo a diretiva *#include*, o próprio analisador busca o nome do arquivo existente entre as *aspas* e adiciona-o no topo da pilha. Além disso, redireciona-se a varredura do código para o início do novo arquivo especificado e salva-se a situação do buffer de entrada no topo de uma pilha denominada de *pilha_buffers*. Essa pilha possui a função de guardar o ponto em que a varredura dos arquivos foi interrompida, a fim de que, futuramente, possa-se retornar a varrer o arquivo logo após o ponto exato onde a diretiva *#include* foi encontrada.

Esse processo de redirecionamento de varredura é aplicado recursivamente para todos os arquivos incluídos, de forma a sempre se ter, no topo de *pilha_arquivos*, o nome

do arquivo sendo analisado pela *CUIA*. Quando a varredura chegar ao fim do arquivo incluído, desempilha-se um elemento, e retorna-se ao arquivo presente no novo topo da pilha, no ponto em que foi anteriormente interrompido. A maneira como o redirecionamento é feito utiliza-se de funções existentes no *Yacc*, tais como a *yy_create_buffer* e *yy_switch_to_buffer*, as quais possuem a capacidade de criar um novo buffer de leitura e de redirecionar a varredura para um novo buffer, respectivamente.

Na Figura 5.3, pode-se observar um exemplo de código C dividido em diferentes arquivos, com diferentes níveis de inclusão. Nesse contexto, os rótulos das setas indicam a ordem de varredura do código pela ferramenta, enquanto que os desenhos ao lado de cada arquivo mostram a situação de *pilha_arquivos* no momento de suas respectivas varreduras.

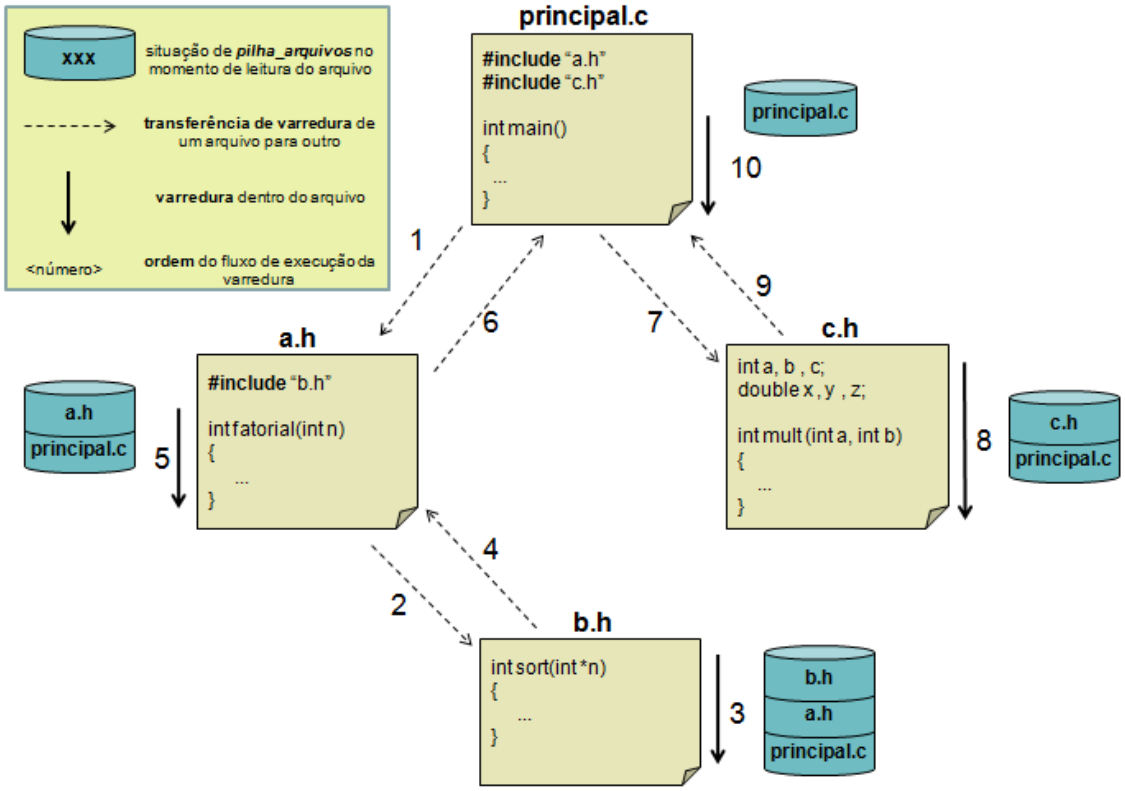

Figura 5.3: Seqüência de Varredura de Código C com Respectiva Situação da Pilha de Arquivos

Em uma declaração de uma variável, em qualquer arquivo do código-fonte, a informação relativa ao nome do arquivo em que ela se encontra estará disponível, sempre, no topo da pilha. Por esse motivo, não é necessária a inclusão de nenhuma ação semântica específica, na gramática, para realizar o controle desta informação, uma vez que todo o seu tratamento é feito pelo analisador léxico. Para isso, foi inserida uma nova expressão regular no analisador léxico, responsável pelo casamento do lexema *#include*. A ação tomada quando esta diretiva é encontrada é a de guardar o nome do arquivo incluído no topo de *pilha_arquivos*, salvar a situação do buffer de entrada em *pilha_buffers* e redirecionar a análise para o início do novo arquivo. Nesse caso, nenhum token é retornado, de forma que o analisador léxico continue o processo de leitura de caracteres de entrada, a fim de que se encontre o próximo token válido para a análise sintática. É importante notar que, quando a varredura do arquivo incluído for finalizada, ocorrerá um término da função *yyparse*, criada pelo *Yacc*. Nesse caso, testa-se o tamanho de *pilha_arquivos*. Se o seu

valor for igual a 1, trata-se de um término de varredura do arquivo principal e, portanto, finaliza-se a análise. Se o valor for maior que 1, é necessário desempilhar um elemento e retornar a análise ao arquivo presente no novo topo da pilha, chamando-se novamente a função *yyparse*. Para que a varredura inicie no ponto onde anteriormente havia sido interrompida, desempilha-se um elemento de *pilha_buffers* e restaura-se o buffer de entrada do arquivo através da chamada à função *yy_switch_to_buffer*.

Para guardar a informação do nome do arquivo em que uma variável é declarada, acrescenta-se um novo comando na ação semântica da listagem 5.7. Esse comando é responsável pela pesquisa do topo de *pilha_arquivos* e o salvamento de tal valor na entrada da tabela de símbolos. Tal modificação pode ser vista na linha 7 da listagem 5.12.

```
1  direct_declarator : IDENTIFIER
2
3    {
4        entrada = criaEntrada();
5        entrada.nome = IDENTIFIER.nome;
6        entrada.tipo = concatena_elementos_fila ( fila_tipos );
7        entrada.arquivo = topo_pilha ( pilha_arquivos );
8
9        /** Insere outras informacoes na entrada **/
10
11       escopo_atual = topo_pilha ( pilha_escopos );
12       entrada.escopo.nome = escopo_atual.nome;
13       entrada.escopo.linha = escopo_atual.linha;
14       entrada.escopo.coluna = escopo_atual.coluna;
15
16       insere_entrada ( escopo_atual.tabela , entrada );
17
18       direct_declarator.entrada = entrada ;
19   }
```

Listagem 5.12: Inserção do Nome do Arquivo na Entrada da Tabela de Símbolos

5.1.5 Obtenção do *Tipo de Array* de uma Variável

Como citado no capítulo 2, uma das informações recuperadas pela *CUIA* é uma *flag* que indica o *tipo de array* de uma variável. Ou seja, a *flag* indica uma das três seguintes possibilidades:

- A variável é um array dinâmico
- A variável é um array estático
- A variável não é um array

A emissão desta informação na saída é possível através da utilização de 3 *flags* distintas, tal como apresentado na tabela a seguir:

Tipo de Array	**Saída**
Array Dinâmico	*DYNAMIC_ARRAY*
Array Estático	*STATIC_ARRAY*
Não é Array	*NOT_ARRAY*

Para este Trabalho, foram utilizadas algumas considerações básicas para realizar a determinação do *tipo de array* da variável. Tais considerações serão explicadas nas subseções 5.1.5.1 e 5.1.5.2. Posteriormente, na subseção 5.1.5.3 será feita a explicação de quais ações foram tomadas para que a informação do *tipo de array* pudesse ser extraída do código-fonte.

5.1.5.1 Arrays Dinâmicos

Considera-se como array dinâmico qualquer array que não possua uma memória previamente alocada pelo compilador. Dessa forma, sua alocação é feita de forma dinâmica, pelo próprio programador, durante a execução do programa. Essa alocação é realizada tipicamente por chamadas à funções externas, como as funções *malloc* ou *calloc* da biblioteca *stdlib.h*.

Para que a alocação de memória para tais variáveis possa ser feita dinamicamente, é necessário que suas declarações sejam feitas através do uso de ponteiros (independente do número de níveis). Dessa forma, pode-se atribuir para tais variáveis o endereço da memória alocada, de modo a permitir o acesso à memória através desses arrays.

Na listagem 5.13, pode-se observar uma declaração típica de um array dinâmico e sua posterior alocação de memória através da chamada à função *malloc*.

```
1 #include <stdlib.h>
2
3 int main()
4 {
5         int *a;
6
7         a = (int*) malloc ( 5 * sizeof(int) );
8         a[3] = 1000;
9 }
```

Listagem 5.13: Exemplo de Declaração e Uso de Array Dinâmico em C

Em nosso Trabalho, considerou-se como array dinâmico, portanto, toda e qualquer variável que possua em sua definição um ou mais níveis de ponteiros. Isso é feito através da adição de asteriscos entre o tipo básico da variável e o seu nome, como pode ser visto na declaração da variável *a* da listagem 5.13. A partir dessa consideração, é importante notar que, mesmo que a alocação de memória não seja realizada no código para tais variáveis, a *flag* indicará que tratam-se de arrays dinâmicos, pois sua declaração contém pelo menos um nível de ponteiro.

5.1.5.2 Arrays Estáticos

Diferentemente dos arrays dinâmicos, os estáticos possuem seus espaços de ocupação em memória previamente alocados pelo compilador. Para que isso seja possível, é necessário que o programador informe o tamanho de cada uma de suas dimensões na própria declaração da variável.

A listagem 5.14 mostra um exemplo de declaração estática de um array e o seu uso pelo programa. Nota-se que não é necessária a realização de alocação de memória pelo programador, uma vez que tal função é executada pelo compilador.

```
1 int main()
2 {
3         int a[100][20];
4
```

```
5           a[0][5] = 2;
6           a[75][15] = 12;
7   }
```

Listagem 5.14: Exemplo de Declaração e Uso de Array Estático em C

No caso da ferramenta *CUIA*, mesmo que o programador informe a existência de uma dimensão, porém não o seu tamanho, considera-se a variável como um array estático. Dessa forma, na lista do tamanho de cada uma das dimensões do array, informa-se que tal dimensão possui um tamanho *indefinido*. Um exemplo de declaração de uma variável com uma dimensão desse tipo pode ser vista abaixo:

int a[][5][4];

5.1.5.3 Ações Tomadas

O reconhecimento de um identificador dentro da declaração de uma variável, por si só, não permite inferir o *tipo de array* a que esta variável pertence. Dessa forma, indica-se, inicialmente, na entrada da tabela de símbolos, que tal variável não é um array. Se forem encontrados tokens que mostrem que, de fato, a variável tratava-se de um array, altera-se a entrada da tabela para que tal informação seje atualizada.

Baseado neste algoritmo, foi acrescentado um novo comando na ação semântica da listagem 5.12, tal como pode ser visto na linha 8 da nova listagem 5.15, responsável pela inicialização do tipo de array da variável.

```
1  direct_declarator : IDENTIFIER
2
3    {
4        entrada = criaEntrada();
5        entrada.nome = IDENTIFIER.nome;
6        entrada.tipo = concatena_elementos_fila ( fila_tipos );
7        entrada.arquivo = topo_pilha ( pilha_arquivos );
8        entrada.tipoArray = NOT_ARRAY;
9
10       /** Insere outras informacoes na entrada **/
11
12       escopo_atual = topo_pilha ( pilha_escopos );
13       entrada.escopo.nome = escopo_atual.nome;
14       entrada.escopo.linha = escopo_atual.linha;
15       entrada.escopo.coluna = escopo_atual.coluna;
16
17       insere_entrada ( escopo_atual.tabela , entrada );
18
19       direct_declarator.entrada = entrada ;
20   }
```

Listagem 5.15: Atualização da Entrada com a Informação *NOT_ARRAY*

A única possibilidade de uma variável ser um array estático é que ela apresente pelo menos uma dimensão informada através de colchetes (exemplo: *int x[3];*). Nesse caso, obrigatoriamente, deverá ser reduzida uma das duas produções seguintes:

$$direct\_declarator \longrightarrow direct\_declarator\ [\ ]$$
$$direct\_declarator \longrightarrow direct\_declarator\ [\ constant\_expression\ ]$$

Considerando esse aspecto, atualiza-se a entrada da tabela com o *tipo de array* estático, quando tais produções forem reduzidas. Tais ações podem ser vistas na listagem 5.16.

```
1  direct_declarator
2    : direct_declarator [ ]
3
4      {
5          direct_declarator[D].entrada.tipoArray = STATIC_ARRAY;
6          direct_declarator[E].entrada = direct_declarator[D].entrada;
7      }
8
9    | direct_declarator [ constant_expression ]
10
11     {
12         direct_declarator[D].entrada.tipoArray = STATIC_ARRAY;
13         direct_declarator[E].entrada = direct_declarator[D].entrada;
14     }
```

Listagem 5.16: Atualização da Entrada com a Informação *STATIC_ARRAY*

Até este momento, pode-se saber duas informações: se uma variável não é array ou se ela é um array estático. A maneira para se descobrir se uma variável é do tipo array dinâmico é realizando-se a verificação da existência de algum nível de ponteiro em sua declaração, que nada mais é que a presença de um ou mais asteriscos entre o seu tipo básico e o nome. Dado um *declarator*, que é o declarador de uma variável, pode-se derivar, a partir dele na gramática, um *direct_declarator* (responsável pelas inferências relativas a STATIC_ARRAY e NOT_ARRAY) ou em um não-terminal *pointer* seguido do não-terminal *direct_declarator*. Esse não-terminal *pointer* é o responsável pela derivação da subárvore sintática que contém cada um dos tokens asterisco da declaração da variável em questão. Dessa forma, criou-se para esse não-terminal um atributo sintetizado denominado de *nivel*, o qual possui a função de guardar a quantidade de níveis de ponteiros que foram varridos pela análise sintática. As ações semânticas acrescentadas para isso estão contidas na listagem 5.17.

```
1  pointer
2    : *
3
4      { pointer.nivel = 1; }
5
6    | * type_qualifier_list
7
8      { pointer.nivel = 1; }
9
10   | * pointer
11
12     { pointer[E].nivel = pointer[D].nivel + 1; }
13
14   | * type_qualifier_list pointer
15
16     { pointer[E].nivel = pointer[D].nivel + 1; }
17   ;
```

Listagem 5.17: Ações Semânticas para a Contagem do Número de Níveis de Ponteiros

Voltando a análise de um não-terminal *declarator*, quando sua produção que deriva em um *pointer* e em um *direct_declarator* for possível de ser reduzida, é garantido que

no atributo sintetizado *nivel* de *pointer* existirá a quantidade de asteriscos da variável em questão e que no atributo sintetizado *entrada* de *direct_declarator* existirá a entrada da tabela de símbolos da mesma variável. Logo, basta acrescentar uma ação semântica que incorpore o valor do atributo *nivel* ao campo *tipo* da entrada da tabela, concatenando-se asteriscos no início da descrição do tipo básico, a fim de formar o tipo final da variável. Essa ação semântica pode ser vista na listagem 5.18. Nota-se que apenas se é informado que o tipo da variável em questão é um DYNAMIC_ARRAY no caso de a variável não ter sido reconhecida como um STATIC_ARRAY anteriormente. Isso se deve ao fato de que é classificada em array estático toda variável que apresentar ambos os níveis de ponteiros e dimensões definidas.

```
1  declarator
2    : pointer direct_declarator
3
4      {
5           if (direct_declarator.entrada.tipoArray != STATIC_ARRAY)
6             direct_declarator.entrada.tipoArray = DYNAMIC_ARRAY;
7
8           direct_declarator.entrada.tipo = concatena_asteriscos ( pointer
                .nivel , direct_declarator.entrada.tipo );
9      }
10
11   | direct_declarator
12   ;
```

Listagem 5.18: Atualização da Entrada com a Informação *DYNAMIC_ARRAY* e o Tipo Final

A visualização de um exemplo de reconhecimento de um tipo de array é mostrado na Figura 5.4, de tal forma que pode-se observar a subárvore sintática originada para a declaração de uma variável com dois niveis de ponteiros e nenhuma dimensão definida. Nesse contexto, a figura ilustra a maneira como são atualizadas as informações do *tipo de array* da variável e do tipo final na entrada da tabela de símbolos.

5.1.6 Obtenção de Informações Extras Relativas a Arrays

Como citado na seção 2.1, existem três informações extras que são coletadas para variáveis do tipo array: o número de dimensões, o tamanho de cada uma delas e a lista de tipos de acesso em comandos *for*. É importante observar que o número de dimensões e o tamanho das mesmas são informações possíveis de serem coletadas apenas para o que denominamos, na subseção 5.1.5.2, de arrays estáticos. Isso se deve ao fato de arrays estáticos definirem dimensões explícitas através de colchetes ([e]), ao contrário de arrays dinâmicos, que são caracterizados pela presença de pelo menos um nivel de ponteiro em sua declaração. O modo como cada uma dessas informações é obtida encontra-se explicado nas subseções seguintes.

5.1.6.1 O Tamanho de Cada Uma das Dimensões do Array

Já que o tratamento do tamanho de cada uma das dimensões de um array destina-se apenas a arrays estáticos, é conveniente presumir que as mesmas duas produções citadas na subseção 5.1.5.3, a qual destinou-se ao detalhamento da identificação de tais tipos de arrays, são também responsáveis pela obtenção das dimensões propriamente ditas.

A primeira produção, como pode-se observar, não possui nenhuma derivação possível de existir entre os colchetes que declaram uma nova dimensão. Nesse caso, a ação

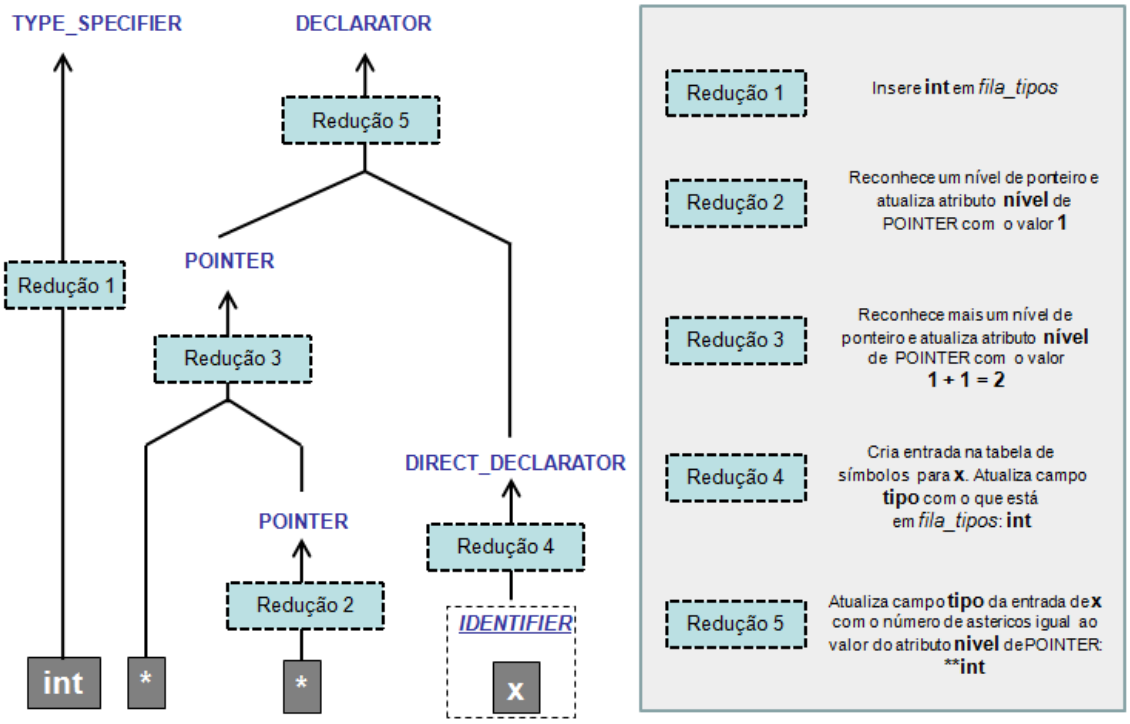

Figura 5.4: Seqüência de Reduções para a Obtenção do Tipo Final de um Array Dinâmico

a ser tomada é a de se adicionar a informação de uma nova dimensão na entrada da tabela de símbolos, com tamanho indefinido. A saída da ferramenta para tal tamanho será igual a -1, a fim de indicar a indefinição da dimensão. Por outro lado, a segunda produção permite que um valor relativo a uma *expressão constante* seja especificado entre os colchetes, de forma que tanto um valor inteiro simples (como 100) como uma expressão aritmética (como 20+100/4) possam ser utilizados para se definir o tamanho da dimensão em questão.

Supõe-se que o resultado final do cálculo da *expressão constante*, que encontra-se presente entre os colchetes de uma dimensão, seja previamente calculado e sintetizado em um atributo denominado de *valor*, no não-terminal *constant_expression*. Dessa forma, no momento da redução dessa segunda produção, pode-se criar a informação de uma nova dimensão na tabela de símbolos com tamanho igual ao valor contido nesse atributo. Para isso, foram adicionados novos comandos nas ações semânticas da listagem 5.16, como pode ser visto na nova listagem 5.19.

```
1  direct_declarator
2    : direct_declarator [ ]
3
4      {
5        direct_declarator[D].entrada.tipoArray = STATIC_ARRAY;
6        dimensao = cria_elemento_lista_dimensoes ( INDEFINIDO );
7        insere_lista ( direct_declarator[D].entrada.listaDimensoes ,
             dimensao );
8        direct_declarator[E].entrada = direct_declarator[D].entrada;
9      }
10
11   | direct_declarator [ constant_expression ]
12
13     {
```

```
14        direct_declarator[D].entrada.tipoArray = STATIC_ARRAY;
15        dimensao = cria_elemento_lista_dimensoes ( constant_expression.
              valor );
16        insere_lista ( direct_declarator[D].entrada.listaDimensoes ,
              dimensao );
17        direct_declarator[E].entrada = direct_declarator[D].entrada;
18      }
```

Listagem 5.19: Ações Semânticas para Reconhecimento do Tamanho das Dimensões

A principal questão a ser tratada, portanto, resume-se ao cálculo do atributo *valor* do não-terminal *constant_expression*. Suas derivações possíveis constituem um conjunto de regras para formar expressões aritméticas e lógicas, a fim de estipular um valor final constante para o tamanho de uma dimensão. A listagem 5.20 mostra alguns exemplos de declarações de arrays com diferentes formatos de definição de tamanhos para suas dimensões.

```
1 int x[4];
2 double y[20][50 + 4];
3 float z[100/4 + 30*2 - 4|2];
4 char w[2][3][4][5][6][7][];
```

Listagem 5.20: Exemplos de Declarações de Arrays Estáticos em C

Uma vez que o processo de varredura do código utiliza-se de um algoritmo *bottom-up*, primeiramente é feito o reconhecimento de cada um dos valores inteiros que compõem a expressão relativa ao tamanho de uma dimensão. Em seguida, são reduzidas as produções que compõem as operações possivelmente contidas na expressão total, como a adição ou a multiplicação. É importante notar que a precedência dos operadores é tratada implicitamente pela própria gramática, já que o algoritmo utilizado pelo *Yacc* funciona de tal forma que uma produção relativa a um operador mais precedente será sempre reduzida antes de uma outra com um operador menos precedente. Por fim, compõe-se o valor final da expressão no atributo *valor*, construindo-o *de baixo para cima* na árvore sintática implícita criada pelo *Yacc*.

Primeiramente, no analisador léxico, no reconhecimento de constantes inteiras, deve-se acrescentar o retorno do valor propriamente dito do token reconhecido. Nesse caso, ao se reconhecer o número 56 como um inteiro, por exemplo, deve-se informar ao analisador sintático não só o reconhecimento de um token CONSTANT_INT, como também do seu valor propriamente dito. O analisador sintático, por conseguinte, passa a ter tal informação disponível nas produções que contém esse tipo de token, podendo manipulá-lo com operadores a fim de formar o valor final. As ações semânticas básicas para tal manipulação estão contidas na listagem 5.21, a fim de ilustrar o modo como é construído o valor final da expressão *de baixo para cima* na árvore sintática. Assim sendo, listam-se apenas algumas das ações, sendo desnecessário ilustrá-las para cada um dos operadores da gramática, visto que são semelhantes. É importante notar que cada um dos não-terminais que são derivados a partir de uma expressão constante contém, também, o atributo *valor*, a fim de guardar os valores parciais do cálculo da expressão.

```
1 primary_expression
2   : CONSTANT_INT    { primary_expression.valor = CONSTANT_INT.valor; }
3
4   ...
5
6 multiplicative_expression
```

```
7    : cast_expression
8
9      { multiplicative_expression.valor = cast_expression.valor; }
10
11   | multiplicative_expression * cast_expression
12
13    { multiplicative_expression[E].valor = multiplicative_expression[D].
         valor * cast_expression.valor; }
14
15   | multiplicative_expression + cast_expression
16
17    { multiplicative_expression[E].valor = multiplicative_expression[D].
         valor + cast_expression.valor; }
18
19   ...
20
21 constant_expression
22   : conditional_expression
23
24     { constant_expression.valor = conditional_expression.valor; }
25
26   ;
```

Listagem 5.21: Ações Semânticas para o Cálculo de uma Expressão Constante

Na listagem 5.21, embora não tenha sido informado por questões de visualização, deve-se observar que o não-terminal *conditional_expression* derivará, em algum passo, em *multiplicative_expression*, da mesma forma que *cast_expression* derivará em *primary_expression*. Dessa forma, constrói-se o valor final da expressão a partir dos terminais do tipo CONSTANT_INT até o atributo *valor* do não-terminal *constant_expression*, tornando possível a adição dos tamanhos de cada uma das dimensões do array na lista existente na entrada da tabela de símbolos. A Figura 5.5 ilustra todo o processo explicado até aqui, de acordo com um exemplo de declaração de array e sua respectiva árvore sintática implícita.

5.1.6.2 O Número de Dimensões do Array

Uma das informações mais simples de ser obtida é o número de dimensões de um array. Conforme foi visto na subseção anterior, criou-se, para cada variável do tipo array estático, uma lista com os tamanhos de cada uma de suas dimensões. Uma maneira simples de se obter o número de dimensões seria, portanto, contando-se o número de elementos dessa lista, ao final da varredura do declarador de tal variável. Optou-se, contudo, em incrementar o campo *numDimensoes* da entrada da tabela de símbolos, após o reconhecimento de cada uma das dimensões. O acréscimo de tal ação pode ser visto na listagem 5.22. Obviamente, o valor deste campo deve ser inicializado com 0, no momento de criação da entrada na tabela.

```
1  direct_declarator
2    : direct_declarator [ ]
3
4      {
5        direct_declarator[D].entrada.tipoArray = STATIC_ARRAY;
6        dimensao = cria_elemento_lista_dimensoes ( INDEFINIDO );
7        insere_lista ( direct_declarator[D].entrada.listaDimensoes ,
            dimensao );
8        direct_declarator[D].entrada.numDimensoes += 1;
```

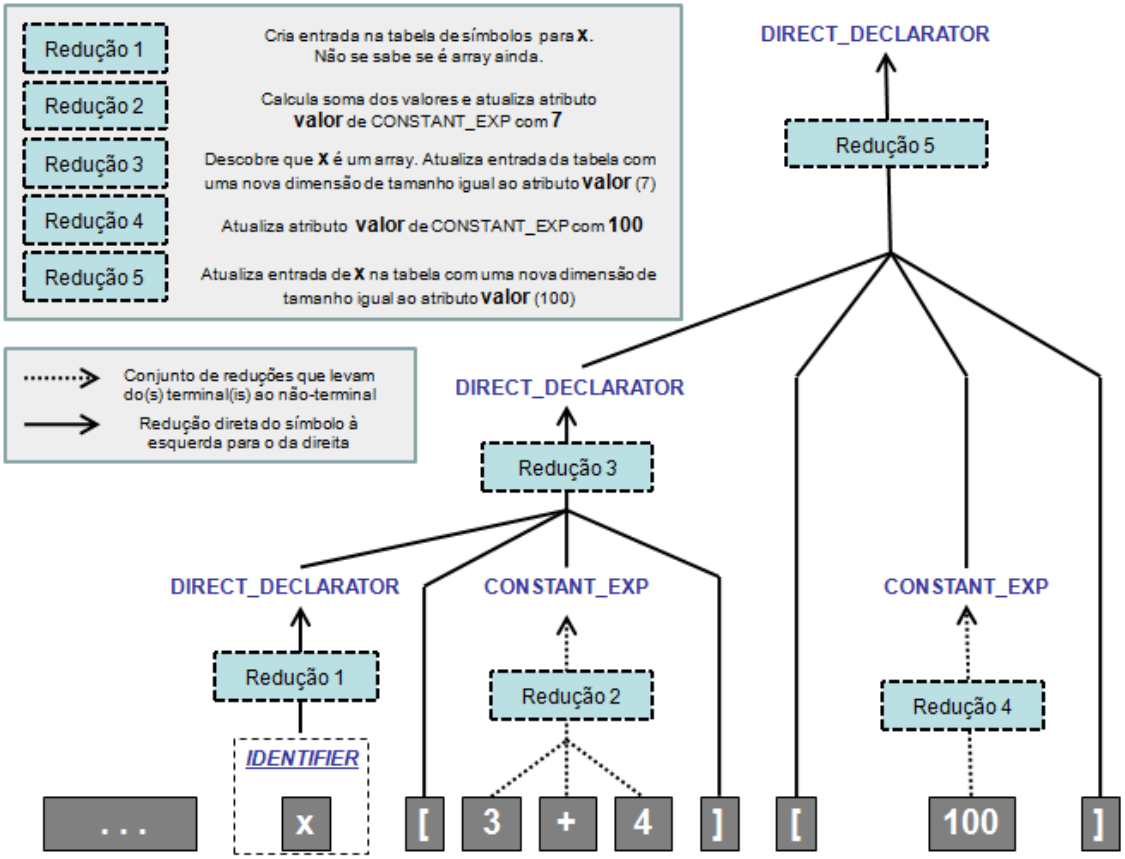

Figura 5.5: Obtenção do Tamanho das Dimensões do Array

```
9         direct_declarator[E].entrada = direct_declarator[D].entrada;
10      }
11
12    | direct_declarator [ constant_expression ]
13
14      {
15        direct_declarator[D].entrada.tipoArray = STATIC_ARRAY;
16        dimensao = cria_elemento_lista_dimensoes ( constant_expression.
              valor );
17        insere_lista ( direct_declarator[D].entrada.listaDimensoes ,
              dimensao );
18        direct_declarator[D].entrada.numDimensoes += 1;
19        direct_declarator[E].entrada = direct_declarator[D].entrada;
20      }
```

Listagem 5.22: Ações Semânticas para a Contagem do Número de Dimensões de um Array

5.1.6.3 *Obtenção dos Tipos de Acesso do Array em Comandos* for

As questões de afinidade de memória em arquiteturas *NUMA* estão diretamente ligadas às políticas de alocação de memória aplicadas a arrays. Nesse caso, uma das informações importantes consideradas em (STANGHERLINI et al., 2010) é o tipo de acesso dessas variáveis em comandos *for*. Uma vez que uma variável do tipo array pode ser acessada em mais de um comando *for* no código-fonte, é necessário obter-se, na verdade, uma lista com os *N* tipos de acesso realizados nos *N* comandos *for* em que a variável é

acessada.

Para que cada tipo de acesso seja relacionado corretamente ao escopo do comando *for* a que pertence, é necessária a criação de uma pilha, denominada de *pilha_fors*, que sempre mantenha em seu topo o contexto do comando *for* em que a varredura se encontra, em um determinado momento. Nesse caso, se o processo de varredura do código-fonte não se encontrar dentro do escopo de nenhum comando deste tipo, a pilha estará vazia. Por outro lado, se existirem comandos *for* aninhados, o topo da pilha conterá o contexto do último comando, apenas. Essa abordagem é semelhante àquelas utilizada para a obtenção do escopo e do nome do arquivo, justificando, novamente, o uso de uma pilha para tal procedimento.

A primeira ação a ser tomada, partindo dos princípios mencionados, é a da manipulação de *pilha_fors*. Assim sendo, logo após o reconhecimento de um token *FOR*, deve-se empilhar um descritor nesta pilha, salvando-se o número da linha e da coluna em que o token aparece no código-fonte. O mesmo descritor deve ser desempilhado após a leitura de toda a estrutura sintática de um comando *for*, uma vez que o seu respectivo escopo deixa de existir. Essas duas ações semânticas podem ser vistas na listagem 5.23.

```
1  iteration_statement
2    : FOR (
3
4      {
5        descritor = cria_elemento_for ( FOR.linha , FOR.coluna );
6        empilha ( pilha_fors , descritor );
7      }
8
9      for_statement
10   ;
11
12 for_statement
13   : expression_statement expression_statement ) statement
14
15     {
16       desempilha ( pilha_fors );
17     }
18
19   | expression_statement expression_statement expression ) statement
20
21     {
22       desempilha ( pilha_fors );
23     }
24
25   ;
```

Listagem 5.23: Ações Semânticas para Manipulação de *pilha_fors*

O não-terminal *statement*, presente nas linhas 10 e 16, é a raiz das derivações possíveis para todos os comandos que podem realizar acessos a variáveis do tipo array, dentro do comando *for*. O fato de se desempilhar o descritor do comando de iteração logo após esse não-terminal garante que, durante todo o processo de varredura do código pertencente ao escopo do *for*, o topo de *pilha_fors* conterá o descritor correto para consulta.

O segundo passo a ser tomado é o do cálculo dos tipos de acesso propriamente ditos. Para isso, criou-se três valores inteiros para representar os acessos de leitura, escrita e leitura/escrita, como listados abaixo (o valor 0 é considerado indefinido):

- Acesso de leitura (R): Valor 1 (em binário: 00000001)

- Acesso de escrita (W): Valor 2 (em binário: 00000010)
- Acesso de leitura/escrita (RW): Valor 3 (em binário: 00000011)

Nota-se que o valor de leitura/escrita é exatamente o resultado da aplicação do operador lógico *OR* entre o valor de leitura e o de escrita. Essa decisão foi tomada pois, dentro do escopo de um comando *for*, diversos comandos com diferentes acessos podem ocorrer com a mesma variável. A aplicação do operador *OR* entre esses diferentes acessos gera, portanto, o tipo de acesso feito dentro do escopo do comando de iteração.

O reconhecimento do identificador de uma variável que é acessada dentro de uma expressão, seja na forma de escrita ou na forma de leitura, obrigatoriamente fará com que a análise sintática reduza a seguinte produção:

$$primary\_expression \longrightarrow IDENTIFIER$$

No momento em que essa produção é reduzida, sabe-se que a variável relativa ao token *IDENTIFIER* está sendo acessada de alguma maneira. Porém, o tipo específico do acesso ainda não é conhecido pela análise sintática. A ação a ser tomada, nesse caso, é a de se verificar se o identificador está presente dentro do escopo de um comando *for*, consultando-se o topo de *pilha_fors*. Se a pilha estiver vazia, nenhuma ação extra é necessária, pois a varredura não se encontra dentro de um comando *for*. Por outro lado, se a pilha não estiver vazia, deve-se verificar, ainda, se o identificador trata-se de um array. Essa verificação pode ser feita pesquisando-se a sua entrada na tabela de símbolos. Caso o identificador não seja um array, nenhuma ação extra é necessária. Caso contrário, a situação encontrada é a de um acesso a um array dentro de um escopo válido, e deve-se criar um novo elemento na lista de tipos de acesso para essa variável. Essas considerações podem ser vistas na listagem 5.24.

```
1  primary_expression
2    : IDENTIFIER
3
4      {
5        entrada = procura_entrada ( pilha_escopos , IDENTIFIER.nome );
6
7        if ( !pilha_vazia(pilha_fors) && entrada.tipoArray != NOT_ARRAY )
8        {
9          descritor_for = topo ( pilha_fors );
10         arquivo_atual = topo ( pilha_arquivos );
11         linha_for = descritor_for.linha;
12         coluna_for = descritor_for.coluna;
13
14         if ( !existe_elemento_lista ( entrada.listaAcessos , linha_for
               , coluna_for , arquivo_atual ) )
15         {
16           descritor_acesso = cria_elemento_lista_acessos ( 0 ,
                 linha_for , coluna_for , arquivo_atual );
17           insere_lista ( entrada.listaAcessos , descritor_acesso );
18         }
19       }
20
21       primary_expression.entrada = entrada;
22     }
```

Listagem 5.24: Ações Semânticas para Manipulação de *pilha_fors*

Sobre a listagem citada, é importante destacar que a função *procura_entrada* não foi listada no capítulo 4 propositalmente, uma vez que naquele momento não haviam esclarecimentos suficientes para citá-la. O funcionamento dela é baseado na procura da entrada relativa ao identificador nas tabelas existentes em *pilha_escopos*. Essa procura é feita dessa maneira pois pode-se estar sendo acessada uma variável de um escopo mais externo ao escopo atual da varredura. Nesse caso, a procura primeiramente é feita na tabela do escopo presente no topo da pilha. Se a entrada não for encontrada, passa-se a procurá-la na tabela do escopo logo abaixo do topo da pilha, e assim sucessivamente. Outro ponto a ser mencionado sobre a ação semântica da listagem é que, no momento de criação do novo elemento da lista de acessos, já são preenchidos os campos que guardam os números da linha e da coluna do comando *for* e o nome do arquivo em que o acesso é realizado. Por fim, a entrada relativa ao identificador é repassada como o atributo *entrada* para o não-terminal superior, a fim de que ela possa ser acessada futuramente pela análise. Dessa forma, todos os não-terminais relativos à expressões possuem esse atributo, o qual é repassado *de baixo para cima* na árvore sintática.

O único dado faltante a ser obtido, a partir desse momento, é o valor do tipo do acesso propriamente dito. Basicamente, a idéia central é atualizar o elemento criado dessa lista com o valor adequado do tipo de acesso, conforme o processo de análise sintática for reduzindo as produções. O modo em que a atualização desse elemento deve ser feita é aplicando-se, nos momentos apropriados, o operador lógico *OR* ao valor de acesso, a fim de incorporar os tipos de leitura ou escrita ao mesmo. O motivo pelo qual foi utilizado esse operador é a possibilidade de se poder *somar* tipos de acesso, ao invés de simplesmente substituí-los por novos, quando encontrados.

Primeiramente, na listagem 5.25, serão mostradas as ações semânticas utilizadas para a atualização do elemento criado da lista com o tipo de acesso de escrita (**W**). A função *atualiza_elemento_lista* também não havia sido listada no capítulo 4, por questões didáticas deste Trabalho. A sua funcionalidade reside, num primeiro momento, em encontrar o elemento da lista de acessos que represente o acesso no comando *for* que está sendo tratado. Ou seja, procura-se na lista aquele elemento que possui os números de linha, coluna e nome do arquivo iguais aos valores presentes nos topos de *pilha_fors* e *pilha_arquivos*. Encontrado o elemento correto, é aplicado o operador *OR* entre o tipo de acesso do elemento e o valor do segundo parâmetro passado para a função.

Sobre a listagem em si, deve-se notar que os terminais *INC_OP* e *DEC_OP* são os operadores responsáveis pelo incremento e decremento de uma variável. Além disso, a produção originada pelo não-terminal *assignment_expression* indica uma expressão na qual um valor é escrito na variável referenciada no símbolo *unary_expression*, presente no lado direito dessa produção.

```
1  postfix_expression
2    : postfix_expression INC_OP
3
4      {
5        listaAcessos = postfix_expression[D].entrada.listaAcessos;
6        atualiza_elemento_lista (listaAcessos , W);
7        postfix_expression[E].entrada = postfix_expression[D].entrada;
8      }
9
10   | postfix_expression DEC_OP
11
12     {
13       listaAcessos = postfix_expression[D].entrada.listaAcessos;
```

```
14        atualiza_elemento_lista (listaAcessos , W);
15        postfix_expression[E].entrada = postfix_expression[D].entrada;
16      }
17    ;
18
19  unary_expression
20    : INC_OP unary_expression
21
22      {
23        listaAcessos = unary_expression[D].entrada.listaAcessos;
24        atualiza_elemento_lista (listaAcessos , W);
25        unary_expression[E].entrada = unary_expression[D].entrada;
26      }
27
28    | DEC_OP unary_expression
29
30      {
31        listaAcessos = unary_expression[D].entrada.listaAcessos;
32        atualiza_elemento_lista (listaAcessos , W);
33        unary_expression[E].entrada = unary_expression[D].entrada;
34      }
35    ;
36
37  assignment_expression
38    : unary_expression assignment_operator assignment_expression
39
40      {
41        listaAcessos = unary_expression.entrada.listaAcessos;
42        atualiza_elemento_lista (listaAcessos , W);
43        assignment_expression[E].entrada = unary_expression.entrada;
44      }
```

Listagem 5.25: Ações Semânticas para Reconhecimento de Acesso de Escrita

É importante notar que as ações semânticas da listagem 5.25 são realizadas mesmo que a variável inicialmente reconhecida no acesso não seja um array ou não pertença ao escopo de um comando *for*. A garantia, contudo, de que o processo funciona, reside no fato de que a função *atualiza_elemento_lista* apenas atualizará o campo de um acesso se o mesmo, de fato, existir na lista de tipos de acesso da variável em questão. Para variáveis que não são arrays ou que não estejam sendo acessadas dentro de um comando de iteração, nenhuma ação é realizada dentro dessa função, uma vez que o elemento a ser atualizado simplesmente não existirá.

Para os acessos do tipo leitura (**R**), o tratamento torna-se mais simples. Uma vez que todos os operadores de leitura possíveis de serem aplicados a variáveis devem obrigatoriamente reduzir uma das quatro produções presentes na listagem 5.26, não é necessária nenhuma ação semântica extra além das expostas nessa listagem. Nota-se, também, que não é necessário repassar a entrada para o não-terminal superior dessas produções, pois nenhum acesso de escrita poderá ser encontrado na parte superior da árvore sintática, a partir deste ponto.

```
1   multiplicative_expression
2     : cast_expression
3
4       {
5         atualiza_elemento_lista(cast_expression.entrada.listaAcessos, R);
6       }
```

```
7
8     | multiplicative_expression * cast_expression
9
10      {
11        atualiza_elemento_lista(cast_expression.entrada.listaAcessos, R);
12      }
13
14    | multiplicative_expression / cast_expression
15
16      {
17        atualiza_elemento_lista(cast_expression.entrada.listaAcessos, R);
18      }
19
20    | multiplicative_expression % cast_expression
21
22      {
23        atualiza_elemento_lista(cast_expression.entrada.listaAcessos, R);
24      }
```

Listagem 5.26: Ações Semânticas para o Reconhecimento de Acesso de Leitura

Esse conjunto de ações realizadas garantem que os tipos de acesso sejam corretamente construídos e atualizados ao longo da análise sintática.

5.2 O Tratamento das Diretivas de Pré-Processamento

Uma questão a ser considerada pela ferramenta é a possibilidade de o usuário utilizar-se de diretivas de pré-processamento tais como a *#include* e a *#define*. Uma vez que essas diretivas não pertencem ao conjunto de comandos da linguagem C, elas não são contempladas pela gramática utilizada para este Trabalho, impossibilitando seu tratamento em nível sintático. Porém, pode-se trabalhar com tais diretivas em nível léxico, acrescentando-se novas expressões regulares que realizem o casamento dessas diretivas e ações que as tratem adequadamente.

A diretiva *#include*, como foi visto na subseção 5.1.4, foi tratada de maneira a percorrer cada um dos caracteres relativos a ela, sem retornar nenhum token a análise sintática. Obtendo-se o nome do arquivo incluído, redirecionava-se a varredura para o mesmo, e salvava-se o contexto em uma pilha. A diretiva *#define*, por outro lado, requer uma análise um pouco mais detalhada, uma vez que a sua definição de nomes pode ser necessária para a coleta de informações. A listagem 5.27 mostra um código-exemplo C em que define-se um valor *N* como sendo igual a 100 e, em seguida, utiliza-se tal valor para especificar o tamanho da dimensão de um vetor. Nesse caso, o analisador léxico, no momento de varredura da declaração do array, deve reconhecer que *N* trata-se de um valor inteiro, e não de um identificador. Caso contrário, a ferramenta indicaria um erro de sintaxe e o processamento seria interrompido, sem que as informações fossem corretamente coletadas.

```
1  #define N 100
2
3  int main()
4  {
5      int vetor[N];
6      int i;
7
8      for (i = 0; i < N; i++)
```

```
9           vetor[i] = i;
10  }
```

Listagem 5.27: Exemplo de Utilização da Diretiva *#define* em C

Para que o analisador léxico reconheça que alguns lexemas são nomes previamente definidos pela diretiva *#define*, é necessário que tal informação seja anteriormente salva em algum lugar. Com esse objetivo, criou-se uma tabela de símbolos específica para guardar entradas que façam a relação entre um nome definido e a sua definição propriamente dita: *tabela_definicoes*. O procedimento adotado quando uma diretiva deste tipo é encontrada, portanto, é o de inserir uma nova entrada nessa tabela, que guarde a definição obtida. No exemplo da listagem 5.27, a ferramenta, ao encontrar a diretiva *#define*, percorreria os demais caracteres do código-fonte e inseriria uma entrada em *tabela_definicoes* com a especificação de que *N* trata-se, na verdade, do valor inteiro 100.

Tendo-se esse armazenamento de definições realizado, a ferramenta *CUIA* tem a capacidade de fazer a diferenciação entre identificadores e definições em nível léxico. Partindo-se desse princípio, no momento que o analisador léxico reconhecer o padrão relativo a um identificador, deve-se verificar se o lexema reconhecido encontra-se em *tabela_definicoes*. Caso não exista, retorna-se o token *IDENTIFIER* para a análise sintática e segue-se a varredura do código-fonte. Porém, se o lexema existir na tabela, coloca-se a sua respectiva definição na entrada, com os caracteres de trás para a frente. Para que esse procedimento seja melhor compreendido, analisaremos novamente o exemplo da listagem 5.27. Quando o analisador léxico encontrar o lexema *N* na declaração do array *vetor*, ele constatará que tal *string* existe em *tabela_definicoes*. Nesse caso, deve-se colocar o *string* 100 na entrada, para que o analisador léxico possa lê-lo e reconhecê-lo como um inteiro. A maneira como isso deve ser feito é colocando-se, no buffer da entrada, os caracteres na ordem inversa: '0', '0' e '1'. Isso garante que o primeiro caractere a ser lido pelo analisador léxico, em seguida, será o '1', garantindo a ordem correta de varredura. A função utilizada para colocar caracteres dentro do buffer de entrada é a *unput*. O analisador sintático, nesse caso, terá como token retornado um inteiro, com valor igual a 100. Ele desconhecerá o fato de que havia um lexema relativo ao padrão identificador, naquele lugar específico do código.

É importante notar que esse procedimento adotado permite a utilização, inclusive, de definições aninhadas, como pode ser visto no exemplo da listagem 5.28. O valor de *N*, nesse caso, deverá ser igual a 15 e, portanto, o tamanho da dimensão do array *vetor* deverá ser 15 também.

```
1  #define A 10
2  #define N A + 5
3
4  int main()
5  {
6      int vetor[N];
7
8      vetor[A - 1] = N;
9  }
```

Listagem 5.28: Exemplo de Utilização de Diretivas *#define* Aninhadas em C

Embora o comando *typedef* não seja uma diretiva de pré-processamento, é importante destacar que o seu tratamento é similar a da diretiva *#define*. Isso se deve ao fato de que comandos *typedef*, em nível léxico, recolhem as definições e as salvam em uma tabela denominada de *tabela_typdefs*. Nesse caso, quando reconhece-se o lexema relativo ao padrão identificador, verifica-se se ele não se encontra tanto em *tabela_definicoes* como

em *tabela_typedefs*. Se não for encontrado em nenhuma das duas, retorna-se o token *IDENTIFIER* para a análise sintática. Caso o lexema seja encontrado em *tabela_typedefs*, retorna-se o token *TYPE_NAME*, alterando-se semanticamente a análise para que ela entre em acordo com a gramática utilizada.

As demais diretivas da linguagem C e do padrão OpenMP, por exemplo, não são importantes para a obtenção de informações pela ferramenta *CUIA*, exigindo tratamentos mais simples do analisador léxico, como o simples descarte das diretivas e caracteres envolvidos.

6 A VALIDAÇÃO DA FERRAMENTA

A validação da ferramenta *CUIA* foi realizada através de sua execução com códigos de benchmarks e aplicações científicas, pelo próprio projeto *MApp*. Nesse capítulo serão indicados os códigos utilizados para essa validação e as restrições existentes no uso da ferramenta *CUIA*.

6.1 Os Benchmarks e a Aplicação de Teste

Além de testes simples e aplicações pequenas usadas para validar a ferramenta *CUIA*, foram utilizados códigos reais de aplicações e benchmarks como o benchmark *Stream* (MCCALPIN, 1991–2007), o *NAS Parallel Benchmarks* (JIN; FRUMKIN, 2010) e a aplicação para simulação de propagação de ondas sísmicas, denominada de *Ondes 3D* (DUPROS et al., 2008). Observa-se que esses mesmos códigos utilizados como teste foram também avaliados pelo pré-processador do projeto *MApp*, com o intuito de verificar o desempenho das aplicações conforme a utilização de diferentes políticas de memória em arquiteturas *NUMA*.

A função do benchmark *Stream* é a de avaliar o desempenho de memória. Para realizar essa avaliação, o benchmark usa três vetores de escopo global e quatro operações diferentes sobre esses vetores (STANGHERLINI et al., 2010). Já o *NAS Parallel Benchmarks* é um benchmark bem conhecido derivado do código de Fluidodinâmica Computacional. Desse benchmark, selecionou-se a aplicação LU e o kernel CG para os testes com a ferramenta. É importante notar que tais benchmarks e aplicações possuem uma quantidade bastante elevada de variáveis e acessos à arrays. Tipicamente, aplicações que utilizam-se de OpenMP tentam paralelizar segmentos de código onde comandos *for* são utilizados em conjunto com manipulações de vetores. Essa paralelização está diretamente ligada ao processo de alocação de memória, o que justifica o enfoque da obtenção dos tipos de acesso da ferramenta *CUIA* apenas dentro do escopo desse tipo de comando.

A aplicação *Ondes 3D* realiza a simulação de propagação de ondas sísmicas e foi desenvolvido pelo *French Geological Survey*. Basicamente, ele é constituído de três etapas: a alocação de dados, a inicialização e o cálculo da propagação. Nessa aplicação, a complexidade de desenvolvimento utiliza-se de arrays dinamicamente alocados, que são acessados de maneira regular e geralmente do tipo *escrita*.

Os códigos dos benchmarks e aplicações fornecidos como entrada à ferramenta *CUIA* e citados nessa seção tiveram suas informações corretamente coletadas e emitidas na saída.

6.2 As Restrições de Uso

Nem todas as corretas funcionalidades puderam ser trabalhadas nessa etapa do desenvolvimento da *CUIA*. A seguir, explicaremos as restrições de uso da ferramenta.

6.2.1 Os Tamanhos das Dimensões

Uma vez que o tratamento das dimensões utilizou-se apenas de valores inteiros e de expressões aritméticas que os manipulassem, a ferramenta não reconhece tamanhos pelo valor de um identificador. Um exemplo de uma situação desse tipo pode ser visto na listagem 6.1. A funcionalidade do código não é importante neste caso. Observa-se que a ferramenta não pode inferir o tamanho do array *a*, e emitirá na saída o valor *-1* para essa dimensão, a fim de indicar a indefinição encontrada. Nota-se que o valor do identificador *N* poderia ter sido informado por uma entrada do usuário, o que também tornaria impossível a determinação do tamanho da dimensão dessa variável.

```
1  int main()
2  {
3          int N;
4          int C = 2;
5
6          N = 10 + C;
7
8          for (C = 0; C < 10; C++)
9          {
10                 int a[N];
11                 a[C] = C;
12         }
13 }
```

Listagem 6.1: Exemplo de Definição de Tamanho de Array com Valor de Identificador, em C

6.2.2 Utilização do *#define*

As possibilidades de uso da diretiva de pré-processamento *#define* consideram o seu uso apenas para definições diretas, tais como as presentes na listagem 6.2.

```
1  #define A 50
2  #define B 20 % 5 + (2 | 3)
3  #define C "teste"
```

Listagem 6.2: Exemplos de Utilizações Aceitas para a Diretiva *#define*

Porém, o uso da diretiva *#define* para usos maiores, como a própria definição de um arquivo (listagem 6.3), não é suportada pela ferramenta. Nesses casos, a simples adição de um comentário na linha relativa ao *#define* pode solucionar o problema.

```
1  #ifndef _ARQUIVO_H_
2  #define _ARQUIVO_H_
3
4  int funcao (int a, int b);
5  int funcao2 (int a, int b);
6
7  #endif
```

Listagem 6.3: Exemplo de Utilização da Diretiva *#define* Não Aceita pela Ferramenta

6.2.3 As Estruturas e Enumerções

Enumerações e estruturas não foram devidamente tratadas, durante a etapa atual de desenvolvimento da ferramenta. O tratamento de tais estruturas são importantes, pois as mesmas podem ser utilizadas como informações úteis pela ferramenta *CUIA*, seja na determinação do tipo da variável ou o tamanho das dimensões de um array.

7 CONCLUSÕES

Este Trabalho apresentou o processo de desenvolvimento de *CUIA - Uma Ferramenta para a Obtenção de Informações de Variáveis em Códigos C*. Para isso, foram explicadas os conceitos básicos de análise, as ferramentas auxiliares utilizadas, as estruturas de dados necessárias e o formato de saída da mesma.

Primeiramente, partindo-se da idéia de se realizar a varredura de um código-fonte escrito na linguagem de programação C, tornou-se necessário algum mecanismo de análise léxica e sintática para o desenvolvimento da *CUIA*. Para isso, foram pesquisadas ferramentas que automatizassem o processo de construção de tais analisadores, sem que fosse necessário programá-los diretamente - tarefa desnecessária e suscetível a erros. Algumas alternativas foram analisadas, escolhendo-se, por fim, o *Lex* e o *Yacc* como as ferramentas responsáveis por construir o analisador léxico e sintático da *CUIA*, respectivamente.

Ambas as ferramentas auxiliares utilizadas requerem uma especificação bem definida da análise que devem construir. Nesse caso, forneceu-se ao *Lex* um conjunto de expressões regulares e ações que identificassem tokens no código e tomassem providências quando alguns segmentos fossem varridos, como as diretivas de pré-processamento. Por outro lado, para o *Yacc* necessitou-se prover uma gramática que informasse a estrutura sintática da linguagem C. A chave para a coleta das informações, como pôde-se observar, é a incorporação de ações semânticas à gramática, de forma a coletar os dados das variáveis simultaneamente ao processo de reconhecimento sintático.

A motivação principal para a criação da *CUIA* baseou-se na necessidade de incorporação de um *parser* ao projeto *MApp* como um módulo independente. Nesse projeto, as informações das variáveis são importantes, pois elas são críticas para a decisão de quais políticas de memória devem ser escolhidas para o controle da afinidade de memória em aplicações C, com OpenMP, sobre arquiteturas *NUMA*. Embora a ferramenta tenha sido utilizada como um módulo do projeto *MApp*, sua forma de utilização e seu formato textual de saída possibilita o uso das informações coletadas em códigos C por quaisquer usuários e ferramentas externas, tornando-a independente de contexto.

Como projeto futuro para a ferramenta desenvolvida, deseja-se trabalhar com outros formatos de saída, que não o textual, e incorporar a implementação de algumas questões não tratadas atualmente, como as enumerações e as estruturas. Além disso, pretende-se coletar outras informações, como o tipo de acesso regular ou irregular de arrays no código.

REFERÊNCIAS

AHO, A.; SETHI, R.; ULLMAN, J. **Compiladores**: princípios, técnicas e ferramentas. 8ª edição.ed. [S.l.]: Guanabara Koogan S.A., 1995.

CARISSIMI, A. et al. Aspectos de Programação Paralela em arquiteturas NUMA. **VIII Workshop em Sistemas Computacionais de Alto Desempenho**, [S.l.], 2007.

DUPROS, F. et al. Exploiting Intensive Multithreading for the Efficient Simulation of 3D Seismic Wave Propagation. **CSE '08: Proceedings of the 11th International Conference on Computational Science and Engineering**, [S.l.], 2008.

HARBISON, S. P.; STEELE, G. L. **C - Manual de Referência**. 1ª edição.ed. [S.l.: s.n.], 2002.

JIN, J. Y. H.; FRUMKIN, M. **The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance, disponível em**: <https://www.nas.nasa.gov/Research/Reports/Techreports/1999/PDF/nas-99-011.pdf>. 2010.

MASON, T.; BROWN, D. **Lex & Yacc**. 2ª edição.ed. [S.l.: s.n.], 1991.

MCCALPIN, J. **Stream**: sustainable memory bandwidth in high performance computers, disponível em: <https://www.nas.nasa.gov/Research/Reports/Techreports/1999/PDF/nas-99-011.pdf>. 1991–2007.

MU, T.; TAO, J.; MCKEE, S. Interactive Locality Optimization on NUMA Architectures. **Software Virtualization**, [S.l.], 2003.

OPENMP. **The openmp specification for parallel programming, disponível em**: <http://www.openmp.org>. 2008.

RIBEIRO, C. et al. Explorando Afinidade em Memória em Arquiteturas NUMA. **WSCAD-SSC**, [S.l.], 2008.

SILBERSCHATZ, A.; GALVIN, P.; GAGNE, G. **Operating System Concepts**. 8ª edição.ed. [S.l.]: John Wiley and Sons, 2009.

STANGHERLINI, I. et al. **Compiling OpenMP Applications to Enhance Memory Affinity on NUMA Machines based on Multi-Core chips**. 2010.

ANEXO A - GRAMÁTICA DA LINGUAGEM C

A sintaxe utilizada nesta gramática é a mesma do *Yacc*. O símbolo inicial é o não-terminal **`translation_unit`**.

```
 primary_expression
    :   IDENTIFIER
    |   CONSTANT
    |   STRING_LITERAL
    |   ( expression )
    ;

postfix_expression
    :   primary_expression
    |   postfix_expression [ expression ]
    |   postfix_expression ( )
    |   postfix_expression ( argument_expression_list )
    |   postfix_expression . IDENTIFIER
    |   postfix_expression PTR_OP IDENTIFIER
    |   postfix_expression INC_OP
    |   postfix_expression DEC_OP
    ;

argument_expression_list
    :   assignment_expression
    |   argument_expression_list , assignment_expression
    ;

unary_expression
    :   postfix_expression
    |   INC_OP unary_expression
    |   DEC_OP unary_expression
    |   unary_operator cast_expression
    |   SIZEOF unary_expression
    |   SIZEOF ( type_name )
    ;

unary_operator
    :   &
    |   *
    |   +
    |   -
    |   ~
    |   !
    ;
```

```
cast_expression
    :   unary_expression
    |   ( type_name ) cast_expression
    ;

multiplicative_expression
    :   cast_expression
    |   multiplicative_expression * cast_expression
    |   multiplicative_expression / cast_expression
    |   multiplicative_expression % cast_expression
    ;


additive_expression
    :   multiplicative_expression
    |   additive_expression + multiplicative_expression
    |   additive_expression - multiplicative_expression
    ;

shift_expression
    :   additive_expression
    |   shift_expression LEFT_OP additive_expression
    |   shift_expression RIGHT_OP additive_expression
    ;

relational_expression
    :   shift_expression
    |   relational_expression < shift_expression
    |   relational_expression > shift_expression
    |   relational_expression LE_OP shift_expression
    |   relational_expression GE_OP shift_expression
    ;

equality_expression
    :   relational_expression
    |   equality_expression EQ_OP relational_expression
    |   equality_expression NE_OP relational_expression
    ;

and_expression
    :   equality_expression
    |   and_expression & equality_expression
    ;

exclusive_or_expression
    :   and_expression
    |   exclusive_or_expression ^ and_expression
    ;

inclusive_or_expression
    :   exclusive_or_expression
    |   inclusive_or_expression | exclusive_or_expression
    ;

logical_and_expression
    :   inclusive_or_expression
    |   logical_and_expression AND_OP inclusive_or_expression
    ;
```

```
logical_or_expression
    :   logical_and_expression
    |   logical_or_expression OR_OP logical_and_expression
    ;

conditional_expression
    :   logical_or_expression
    |   logical_or_expression ? expression : conditional_expression
    ;

assignment_expression
    :   conditional_expression
    |   unary_expression assignment_operator assignment_expression
    ;

assignment_operator
    :   =
    |   MUL_ASSIGN
    |   DIV_ASSIGN
    |   MOD_ASSIGN
    |   ADD_ASSIGN
    |   SUB_ASSIGN
    |   LEFT_ASSIGN
    |   RIGHT_ASSIGN
    |   AND_ASSIGN
    |   XOR_ASSIGN
    |   OR_ASSIGN
    ;

expression
    :   assignment_expression
    |   expression , assignment_expression
    ;

constant_expression
    :   conditional_expression
    ;

declaration
    :   declaration_specifiers ;
    |   declaration_specifiers init_declarator_list ;
    ;

declaration_specifiers
    :   storage_class_specifier
    |   storage_class_specifier declaration_specifiers
    |   type_specifier
    |   type_specifier declaration_specifiers
    |   type_qualifier
    |   type_qualifier declaration_specifiers
    ;

init_declarator_list
    :   init_declarator
    |   init_declarator_list , init_declarator
    ;
```

```
init_declarator
    :   declarator
    |   declarator = initializer
    ;

storage_class_specifier
    :   TYPEDEF
    |   EXTERN
    |   STATIC
    |   AUTO
    |   REGISTER
    ;

type_specifier
    :   VOID
    |   CHAR
    |   SHORT
    |   INT
    |   LONG
    |   FLOAT
    |   DOUBLE
    |   SIGNED
    |   UNSIGNED
    |   struct_or_union_specifier
    |   enum_specifier
    |   TYPE_NAME
    ;

struct_or_union_specifier
    :   struct_or_union IDENTIFIER { struct_declaration_list }
    |   struct_or_union { struct_declaration_list }
    |   struct_or_union IDENTIFIER
    ;

struct_or_union
    :   STRUCT
    |   UNION
    ;

struct_declaration_list
    :   struct_declaration
    |   struct_declaration_list struct_declaration
    ;

struct_declaration
    :   specifier_qualifier_list struct_declarator_list ;
    ;

specifier_qualifier_list
    :   type_specifier specifier_qualifier_list
    |   type_specifier
    |   type_qualifier specifier_qualifier_list
    |   type_qualifier
    ;

struct_declarator_list
    :   struct_declarator
    |   struct_declarator_list , struct_declarator
```

```
    ;

struct_declarator
    :   declarator
    |   : constant_expression
    |   declarator : constant_expression
    ;

enum_specifier
    :   ENUM { enumerator_list }
    |   ENUM IDENTIFIER { enumerator_list }
    |   ENUM IDENTIFIER
    ;

enumerator_list
    :   enumerator
    |   enumerator_list , enumerator
    ;

enumerator
    :   IDENTIFIER
    |   IDENTIFIER = constant_expression
    ;

type_qualifier
    :   CONST
    |   VOLATILE
    ;

declarator
    :   pointer direct_declarator
    |   direct_declarator
    ;

direct_declarator
    :   IDENTIFIER
    |   ( declarator )
    |   direct_declarator [ constant_expression ]
    |   direct_declarator [ ]
    |   direct_declarator ( parameter_type_list )
    |   direct_declarator ( identifier_list )
    |   direct_declarator ( )
    ;

pointer
    :   *
    |   * type_qualifier_list
    |   * pointer
    |   * type_qualifier_list pointer
    ;

type_qualifier_list
    :   type_qualifier
    |   type_qualifier_list type_qualifier
    ;

parameter_type_list
    :   parameter_list
```

```
    |   parameter_list , ELLIPSIS
    ;

parameter_list
    :   parameter_declaration
    |   parameter_list , parameter_declaration
    ;

parameter_declaration
    :   declaration_specifiers declarator
    |   declaration_specifiers abstract_declarator
    |   declaration_specifiers
    ;

identifier_list
    :   IDENTIFIER
    |   identifier_list , IDENTIFIER
    ;

type_name
    :   specifier_qualifier_list
    |   specifier_qualifier_list abstract_declarator
    ;

abstract_declarator
    :   pointer
    |   direct_abstract_declarator
    |   pointer direct_abstract_declarator
    ;

direct_abstract_declarator
    :   ( abstract_declarator )
    |   [ ]
    |   [ constant_expression ]
    |   direct_abstract_declarator [ ]
    |   direct_abstract_declarator [ constant_expression ]
    |   ( )
    |   ( parameter_type_list )
    |   direct_abstract_declarator ( )
    |   direct_abstract_declarator ( parameter_type_list )
    ;

initializer
    :   assignment_expression
    |   { initializer_list }
    |   { initializer_list , }
    ;

initializer_list
    :   initializer
    |   initializer_list , initializer
    ;

statement
    :   labeled_statement
    |   compound_statement
    |   expression_statement
    |   selection_statement
```

```
    |   iteration_statement
    |   jump_statement
    ;

labeled_statement
    :   IDENTIFIER : statement
    |   CASE constant_expression : statement
    |   DEFAULT : statement
    ;

compound_statement
    :   { }
    |   { statement_list }
    |   { declaration_list }
    |   { declaration_list statement_list }
    ;

declaration_list
    :   declaration
    |   declaration_list declaration
    ;

statement_list
    :   statement
    |   statement_list statement
    ;

expression_statement
    :   ;
    |   expression ;
    ;

selection_statement
    :   IF ( expression ) statement
    |   IF ( expression ) statement ELSE statement
    |   SWITCH ( expression ) statement
    ;

iteration_statement
    :   WHILE ( expression ) statement
    |   DO statement WHILE ( expression ) ;
    |   FOR ( expression_statement expression_statement ) statement
    |   FOR ( expression_statement expression_statement expression ) statement
    ;

jump_statement
    :   GOTO IDENTIFIER ;
    |   CONTINUE ;
    |   BREAK ;
    |   RETURN ;
    |   RETURN expression ;
    ;

translation_unit
    :   external_declaration
    |   translation_unit external_declaration
    ;
```

```
external_declaration
    :   function_definition
    |   declaration
    ;

function_definition
    :   declaration_specifiers declarator declaration_list compound_statement
    |   declaration_specifiers declarator compound_statement
    |   declarator declaration_list compound_statement
    |   declarator compound_statement
    ;
```