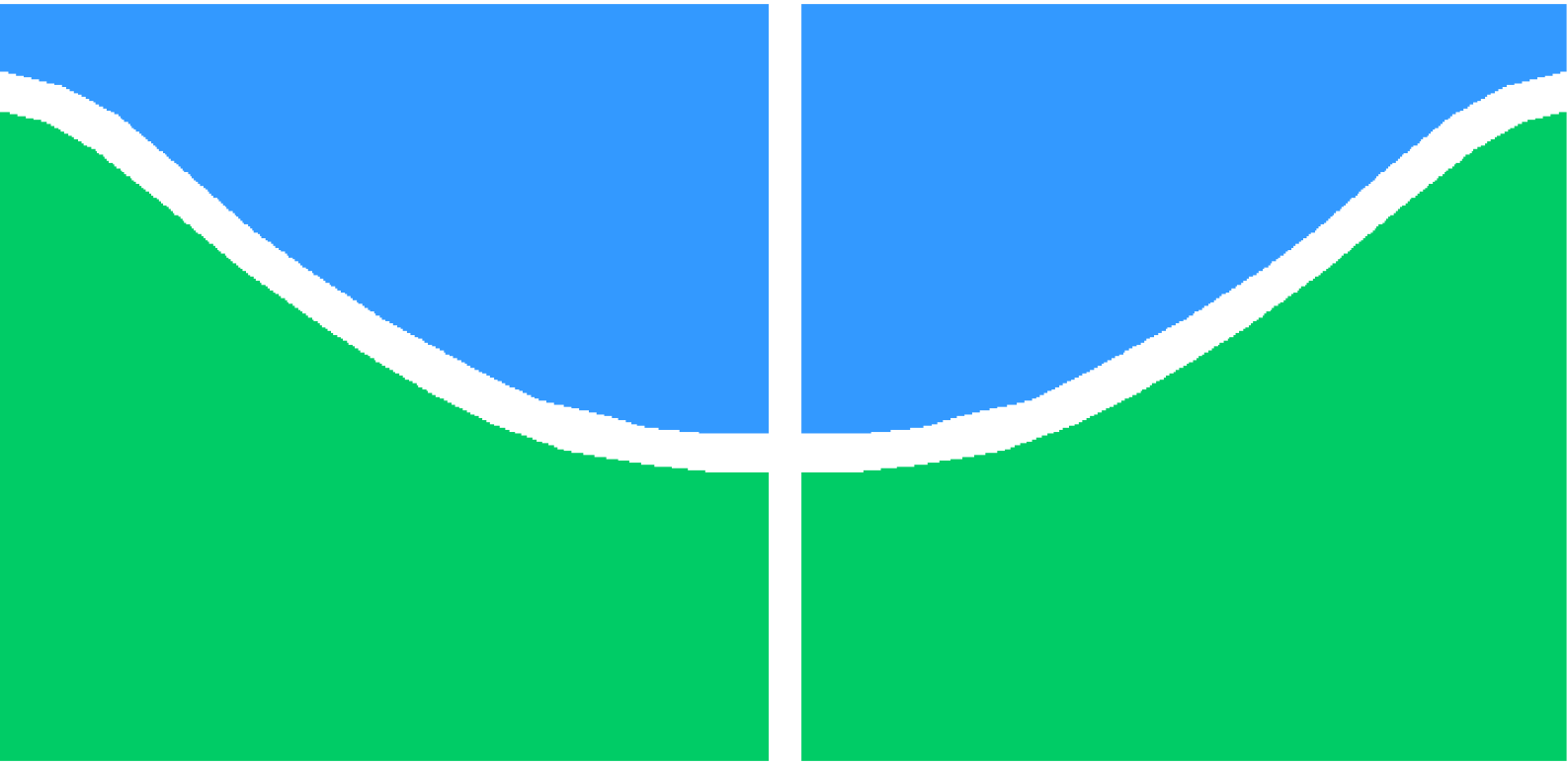

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de Software

Protótipo de Ferramenta para Consulta de Código Fonte

Autor: Charles Daniel de Oliveira

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF

2014

Charles Daniel de Oliveira

Protótipo de Ferramenta para Consulta de Código Fonte

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software .

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF

2014

Charles Daniel de Oliveira

Protótipo de Ferramenta para Consulta de Código Fonte/ Charles Daniel de Oliveira. – Brasília, DF, 2014

94 p.

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. consulta-de-código. 2. análise-estática. 3. *parsing*. I. Prof. Dr. Luiz Augusto Fontes Laranjeira. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Protótipo de Ferramenta para Consulta de Código Fonte

Charles Daniel de Oliveira

Protótipo de Ferramenta para Consulta de Código Fonte

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software .

Trabalho aprovado. Brasília, DF, 5 de dezembro de 2014:

Prof. Dr. Luiz Augusto Fontes Laranjeira
Orientador

Prof. Dr. André Barros de Sales
Membro convidado

Prof. Dr. Sérgio Antônio Andrade de Freitas
Membro convidado

Brasília, DF
2014

Agradecimentos

Agradeço primeiramente à minha mãe, dona Florentina, por apoiar e acreditar no esforço que empreendi durante os 6 anos de faculdade. Em especial o professor e orientador Luiz Augusto Fontes Laranjeira, por ter estabelecido contato cooperativo com pesquisadores do *NIST*, possibilitando a estadia de um ano nos Estados Unidos. Período o qual foram realizadas pesquisas em Análise Estática de Código para segurança, sendo fonte principal de motivação para o desenvolvimento do presente trabalho de conclusão de curso. À Paul E. Black, pesquisador do *NIST* e líder de projeto, por ter paciência e me apresentar aspectos importantes de qualidade de código e desenvolvimento de seguro de software. Agradeço também os professores que fizeram questão de lecionar boas aulas com o objetivo único do aprendizado dos alunos, sem eles a FGA não teria os resultados que tem hoje. Agradeço também aos meus colegas e amigos de graduação, em especial ao colega e amigo Rodrigo Siqueira de Melo, por estar sempre presente e em estilo desbravador, gerando inspiração mútua na dupla ao produzir ótimos trabalhos durante a graduação.

"I have not failed,
I have just found
10,000 ways that won't work".
Thomas A. Edison.

Resumo

Neste trabalho de conclusão de curso será apresentado o desenvolvimento do *ezparser*, um protótipo de ferramenta a ser desenvolvida durante esta disciplina, cujo objetivo principal é a extração de códigos fonte escritos em linguagem *C* para auxiliar no trabalho de ferramentas de análise estática de código. A ferramenta faz uso de um sistema gerenciador de banco de dados para armazenar informações.

Palavras-chaves: consulta de código. análise estática. *parsing*.

Abstract

In this completion of course work will be presented the *ezparser* development, a draft of a tool built during this subject with the main goal of extracting source code written in C programming language in order to act as a helper for static code analysis tools. The tool makes use of a database management system to store information.

Key-words: code querying. static analysis. parsing.

Lista de ilustrações

Lista de tabelas

Lista de códigos

Lista de abreviaturas e siglas

AST	Árvore de sintaxe abstrata (*Abstract Syntax Tree*)
BNF	Forma Backus-Naur (*Backus-Naur Form*)
BOF	Estouro de *buffer* (*Buffer OverFlow*)
CFG	Gramática sem contexto (*Context-Free Grammar*)
CQL	Linguagem de consulta de código (*Code Query Language*)
CWE	Numeração de fraquezas comuns de *software* (*Common Weakness Enumeration*)
DBMS	Sistema gerenciador de banco de dados (*DataBase Management System*)
DSL	Linguagem de domínio específico (*Domain Specific Language*)
NIST	Instituto nacional de normas e tecnologias (*National Istitute of Standards and Techonology*)
NVD	Banco de dados nacional de vulnerabilidades (E.U.A.) (*National Vulnerability Database*)
OWASP	Projeto para segurança em aplicações *web* abertas (*Open Web Application Security Project*)
SCA	Análise estática de código (*Static Code Analysis*)
SQL	Linguagem de consulta estruturada (*Structured Query Language*)

Sumário

1 Introdução

Software consiste em: (1) instruções (programas de computador) que, quando executadas, fornecem características, funções e desempenho desejados; (2) estruturas de dados que possibilitam aos programas manipular informações adequadamente; e (3) informação descritiva, tanto na forma impressa como na virtual, descrevendo a operação e o uso dos programas (PRESSMAN, 2001). Em 1970 menos de 1% da população global era capaz de definir esse termo de maneira inteligente, mas hoje em dia, a maior parte dos profissionais e muitos outros indivíduos do público em geral acham que entendem de *software*. Mas entendem mesmo? (PRESSMAN, 2001). O conceito levou vários anos até se estabelecer na academia, até que no final da década de 80 se deram início as primeiras profissões realmente denominadas *engenheiro de software*. Esse profissional deve estar apto a resolver problemas utilizando-se de técnicas de modelagem, aprendendo com erros e evitando que os mesmos aconteçam no futuro. O engenheiro de *software* também é responsável por escrever código testável, reutilizável e preparado para mudanças (LAPLANTE, 2007).

A partir do código, informações importantes que representam características do *software*, podem ser extraídas e analisadas. A essa análise dá-se o nome de *análise de código*, ramo de estudos que tem como objetivo principal obter informações relevantes a partir do código fonte. Esse conceito pode ser dividido em duas vertentes: análise dinâmica e análise de estática código (SCA). A análise dinâmica é realizada em tempo de execução, em que o programa em questão é testado com dados simulados, muito comum em aplicações *web* (BALZAROTTI et al., 2008). A análise estática de código é a atividade destinada à obtenção de informações a partir de um programa de computador (código fonte ou código objeto). Nesse contexto, o termo estática se remete ao fato de não haver qualquer execução do código sendo analisado (WICHMANN et al., 1995).

Atualmente, existem inúmeros projetos de *software* sendo desenvolvidos e mantidos, cada um podendo chegar a milhões de linhas de código (CODEBASES, 2013). Diferentemente de outros produtos, o programa de computador possui diversas características únicas que, consequentemente, restringem o número de profissionais qualificados para lidar com tais tecnologias, tornando-se dispendiosa e inviável a prática de análise estática manual. Ferramentas de SCA estão ganhando força como técnica complementar à automatização desse processo, obtendo garantia adicional para itens críticos de *software* (WICHMANN et al., 1995).

Durante a análise de sistemas de *software* tem-se o importante desafio de como implementar uma análise em particular para diferentes tipos de linguagem de programação.

Uma solução para esse problema é a criação de uma análise única utilizando-se consulta de código para abstrair as especificidades de cada linguagem sendo analisada. Nos últimos 10 anos muitas tecnologias de consulta de código têm sido desenvolvidas, baseadas em diferentes tipos de formalismo. Cada tecnologia possui sua própria linguagem de consulta e um conjunto específico de funcionalidades (HAGE et al., 2011). É nesse contexto que o presente trabalho será desenvolvido.

1.1 Objetivo geral

Obteve-se ao final deste trabalho, um protótipo de uma ferramenta de consulta de código escrito na linguagem de programação *C*, capaz de prover informações básicas a cerca do código analisado.

1.1.1 Objetivos específicos

Foram estipulados três objetivos específicos para melhor organizar o andamento do projeto e a conclusão do trabalho:

- **Extração de código** - essa é a funcionalidade central do protótipo. Foram utilizadas ferramentas para análise léxica e sintática.
- **Persistência em banco de dados** - o protótipo disponibilizará as informações de código em banco de dados, permitindo consultas de código.
- **Validação do protótipo** - para apresentar o potencial do protótipo, foram consultados dados de um determinado código fonte.

1.2 Metodologia

Para a completude deste trabalho, a metodologia se baseou em: (i) pesquisas bibliográficas para se conhecer o assunto de análise estática de código e suas vertentes; (ii) prototipação de uma ferramenta de consulta de código e; (iii) validação do protótipo submetendo-se códigos e obtendo-se consultas.

1.3 Justificativa

Atualmente, é comum, em projetos de *software*, se chegar às milhares de linhas de código, desenvolvidas por diversos programadores que podem já não fazer parte da equipe. Nesse sentido, este trabalho justifica-se na necessidade de se conhecer o código de um *software*, contribuído por diversas pessoas durante diversos períodos de tempo.

1.4 Organização do trabalho

A introdução deste trabalho visa ganhar a atenção do leitor quanto à análise estática de código, área a ser explicada com maiores detalhes no Capítulo 2; posteriormente, no mesmo capítulo, na Seção 2.5, será apresentada uma subárea da SCA a *consulta de código* e suas utilidades; aprofundando-se tecnicamente, o Capítulo 3 caracteriza as tecnologias utilizadas durante desenvolvimento do *ezparser*, foco principal deste trabalho; o Capítulo 4 destrincha a ferramenta detalhadamente; os resultados e conclusões obtidas foram apresentados no Capítulo 5.

2 Análise estática de código

Neste capítulo foram detalhadas características da análise estática de código, ramo da engenharia de *software* voltado para obter informações acerca de um programa (ANDREY, 2012) a partir do código fonte. Inicialmente, na Seção 2.1, são apresentados termos, não-oficiais, utilizados quando se trata desse tipo de análise, como exemplo, o nome *caso de teste* possui entendimento diferente se comparado ao mesmo termo em outras subáreas da engenharia de *software*. Na Seção 2.2 são mostrados pontos em que a SCA pode auxiliar na qualidade do código de um produto de *software*, gerando métricas determinísticas a partir de determinados trechos do código. Em seguida, na Seção 2.3 são abordados itens em segurança de *software* plausíveis de detecção a partir da SCA.

2.1 Terminologia

A partir do levantamento bibliográfico realizado, não foi encontrada especificação formal de vocabulário para a SCA. Entretanto, é possível notar expressões recorrentes ao realizar leitura de materiais produzidos por organizações como OWASP e NIST. Este trabalho não visa a oficialização dos termos a serem apresentados a seguir. O intuito desta seção é somente apresentar ao leitor qual o significado de tais termos, afim de minimizar conflitos semânticos. Considere o Código 2.1 como exemplo para auxiliar nos conceitos a serem apresentados a seguir.

Lista de códigos 2.1 Exemplo de *warnings*

```
1  char a[10], i = 9, * ptr;
2
3  a[i++]  = 'b';
4  a[10]   = 'c';
5  a[i]    = 'd';
6
7  strcpy(a, "str");
8
9  ptr = a + 9;
10 *++ptr = 'e';
```

Varredura a ferramenta SCA analisa o código especificado. Ao final da varredura é gerado um *report*, com o relatório específico apontando todas as vulnerabilidades encontradas pela ferramenta, em que cada uma delas é dado o nome de *warning*.

Warning são os achados da ferramenta, podendo, ou não, estar caracterizado com o tipo da vulnerabilidade encontrada, o número da linha, o identificador da CWE e ainda uma possível dica de como remover a fraqueza.

Verdadeiro positivo é um *warning* que aponta corretamente uma vulnerabilidade no código. Dado um *report*, somente as linhas 4, 5 e 10 seriam consideradas corretas.

Falso positivo é um *warning* equivocado. Não é muito incomum de encontrar ferramentas que reportem a linha 7 como sendo defeituosa, já que a função *strcpy* (faz cópia dos *bytes* do segundo argumento para o primeiro argumento) é considerada perigosa (WILLIAMS, 2009). Porém, um *warning* para a linha 7 está errado, pois o tamanho da cadeia de caracteres do segundo argumento da função é 4 e o tamanho do vetor '*a*' é 10, ou seja, havendo espaço suficiente para a operação. Apenas ferramentas robustas são capazes de identificar tal linha como não defeituosa.

Falso negativo é um defeito de código não reportado pela ferramenta. A linha 10 é um exemplo de código que muitas ferramentas não conseguem identificar. Resumidamente, o ponteiro '*ptr*' recebe o último endereço do vetor '*a*' na linha 9, em seguida o seu valor é incrementado em uma posição e, por fim, o caractere '*e*' é escrito na posição acima do último *byte* de '*a*', gerando assim o *buffer overflow*. Operações desse tipo geralmente são extremamente difíceis pois envolvem mais variações de código ao mesmo tempo.

Variações de código são elementos da linguagem utilizados. Um laço, condicional, recursão, invocação de função entre outros são exemplos de variações de código. Recursos desse tipo são utilizados em testes sintéticos para simular códigos reais.

Testes sintéticos/casos de teste são códigos gerados com o único fim de serem submetidos às ferramentas. O Código 2.1 é um exemplo de teste sintético. Possuem diversas variações de código para testar a flexibilidade das ferramentas. A vantagem dos testes sintéticos é o conhecimento prévio das linhas defeituosas, facilitando a categorização de verdadeiros positivos, falsos positivos e falsos negativos.

Testes reais são códigos fontes de software reais, usados para avaliar a qualidade da ferramenta em termos de escalabilidade e prática em um projeto real.

2.2 Métricas de código

Nos últmos anos foi possível perceber o aumento significativo do número de projetos cadastrados em plataformas *forge*, espaços na *web* reservados para se manter um projeto de *software*. Tais plataformas geram uma espécie de painel de controle para cada projeto, apresentando o número de arquivos, quantidade de *downloads*, número de desenvolvedores, quais linguagens de programação utilizadas, variando-se de *forge* para *forge*. Um exemplo é o *Github*, que hospeda softwares de diversas linguagens e utiliza o sistema

de controle de versão *Git*. Esse sistema faz o controle das alterações através de *commits*, pequenas modificações no código. O *Github* então faz o uso do número de *commits* de cada projeto como parâmetro de decisão verificando se tal projeto está ou não ativo.

Tendo-se acesso ao código fonte é possível realizar medições sobre a sua estrutura e organização interna de forma a avaliar sua qualidade (MEIRELLES, 2013). Como mostrado anteriormente, métricas têm se tornado padrões de visualização do estado de um projeto de *software*. Informações dessa natureza permitem quantificar a qualidade de um software. A SCA entra neste cenário auxiliando na extração de insumos necessários para métricas de código, sendo uma das mais comuns a *complexidade ciclomática*. Essa parte do princípio que a complexidade depende do número de condições (caminhos), correspondendo ao número máximo de percursos linearmente independentes em um *software*. A proposta é que se possa medir a complexidade do programa, assim orientando o desenvolvimento e os testes do software (MCCABE, 1976; MEIRELLES, 2013).

Uma das motivações principais deste trabalho é dar ao engenheiro de *software* o poder de se extrair diferentes métricas. O protótipo visa explorar a estrutura de sistema de banco de dados para facilitar a combinação das informações extraídas. Com isso, espera-se a criação de novas métricas, que possam trazer maior entendimento e domínio sobre determinado código fonte.

2.3 Segurança

Uma das principais características de um programa de computador é o fato de ser escrito por um ser humano, sendo assim passível de falhas. São inúmeros os casos de desastres causados por falhas de software. Erros ocorrem devido à entradas que causam comportamento não planejado. O problema desse cenário é a incapacidade de se identificar falhas em potencial. Existe um esforço por parte do governo dos Estados Unidos na questão de entendimento de fraquezas de *software* (CHRISTEY et al., 2011).

Uma das principais contribuições realizadas é a criação de um vocabulário comum ao se referir sobre falhas de *software*, a *Common Weakness Enumeration* (CWE) (CHRISTEY et al., 2011). O objetivo da iniciativa é estabelecer um meio comum no qual pesquisadores e interessados do mundo inteiro possam se alinhar conceitualmente ao discutir vulnerabilidades de *software*, definidas como falta de *software* que pode levar a uma falha de segurança no sistema quando executado, de forma sistemática e efetiva.

Uma vez sabido como identificar uma vulnerabilidade, outros agentes do governo americano e de outras partes do mundo centralizaram o registro de falhas de segurança dos principais *softwares* utilizados no mercado, categorizando-se o tipo de falta. Essa base de dados foi chamada de *National Vulnerability Database* (NVD), situado no *National Institute of Standards and Techonology* (NIST). O NIST é uma agência norte-americana

voltada para pesquisas em conjunto com a indústria com o objetivo de se estabelecer padrões e normas para produtos e tecnologias usadas no mercado.

Em pesquisas desenvolvidas pelo NIST, em conjunto com demais agências de apoio e dispondo-se de uma fonte rica de informações organizadas (NVD), foi possível obter uma listagem dos tipos de falhas mais comuns em programas de computador (CHRISTEY et al., 2011). A Tabela 1 ilustra os 25 erros de *software* mais perigosos de 2011, segundo o escopo disponível no NVD.

Rank	***ID***	***Name***
1	CWE-89	*SQL Injection*
2	CWE-78	*OS Command Injection*
3	**CWE-120**	***Classic Buffer Overflow***
4	CWE-79	*Cross-site Scripting*
5	CWE-306	*Missing Authentication for Critical Function*
6	CWE-862	*Missing Authorization*
7	CWE-798	*Use of Hard-coded Credentials*
8	CWE-311	*Missing Encryption of Sensitive Data*
9	CWE-434	*Unrestricted Upload of File with Dangerous Type*
10	CWE-807	*Reliance on Untrusted Inputs in a Security Decision*
11	CWE-250	*Execution with Unnecessary Privileges*
12	CWE-352	*Cross-Site Request Forgery (CSRF)*
13	CWE-22	*Path Traversal*
14	CWE-494	*Download of Code Without Integrity Check*
15	CWE-863	*Incorrect Authorization*
16	CWE-829	*Inclusion of Functionality from Untrusted Control Sphere*
17	CWE-732	*Incorrect Permission Assignment for Critical Resource*
18	CWE-676	*Use of Potentially Dangerous Function*
19	CWE-327	*Use of a Broken or Risky Cryptographic Algorithm*
20	CWE-131	*Incorrect Calculation of Buffer Size*
21	CWE-307	*Improper Restriction of Excessive Authentication Attempts*
22	CWE-601	*URL Redirection to Untrusted Site ('Open Redirect')*
23	CWE-134	*Uncontrolled Format String*
24	CWE-190	*Integer Overflow or Wraparound*
25	CWE-759	*Use of a One-Way Hash without a Salt*

Tabela 1 – Os 25 erros de *software* mais perigosos de 2011

2.3.1 *Buffer overflow*

Buffer overflows (BOF) têm sido a forma mais comum de vulnerabilidade de segurança nos últimos dez anos. Mais além, as vulnerabilidades de BOF são as dominantes na área de invasão remota em redes, onde usuários anônimos da *internet* injetam comandos com o intuito de tomarem controle parcial ou total de uma máquina remota. Por isso, esse tipo de ataque tem sido um dos principais motivos para se preocupar com segurança

(COWAN et al., 2000). Em outas palavras, BOFs são pequenos lembretes da mãe natureza sobre as leis da física que dizem: ao tentar inserir mais conteúdo do que o recipiente é capaz de suportar, haverá comportamento inesperado.

Aplicações em linguagem *C* existem há diversas décadas e BOFs faz parte dessa história por ser muito resistente à extinção. No ano de 2011, esse tipo de vulnerabilidade foi o terceiro erro mais comum reportado conforme mostra a Tabela 1. Nos próximos parágrafos, o BOF, e suas variações, será caracterizado para melhor entendimento do leitor.

Um processo é uma instância de um programa de computador que está sendo executado, contendo informações sobre o código e sobre o estado atual de execução. A Figura 1 ilustra o esquema didático sobre como o processo é iniciado. Seus espaços de memória se dividem em basicamente três principais blocos: *code*, *stack* e *heap*. Em *code* são armazenados todos os códigos de máquina referentes àquele processo, por exemplo, laços de repetição, estruturas condicionais, funções, entre outros (TANENBAUM, 2007). Considere o Código 2.2 para a explicação a seguir. O *stack* guarda informações sobre chamadas e retornos de funções, nele também são armazenados os dados declarados estaticamente em tempo de compilação, como por exemplo o vetor da linha 2 e cadeia de caracteres da linha 3. Por fim, localizam-se no *heap* blocos de memória alocados alocados em tempo de execução, como, por exemplo, o vetor da linha 6 e cadeia de caracteres da linha 7.

Lista de códigos 2.2 Exemplo de alocação de memória no *heap* e no *stack*

```
1 /* Exemplo de alocação de memória no stack */
2 int  a[10]; /* aloca 10 inteiros */
3 char a[] = "123456789"; /* aloca 10 caracteres */
4
5 /* Exemplo de alocação de memória no heap */
6 int  * a = malloc(sizeof(int)); /* aloca 10 inteiros */
7 char * a = malloc(sizeof(char) * 10); /* aloca 10 caracteres*/
```

Uma vez definidos os conceitos de memória *stack* e *heap* é possível introduzir o significado de *buffer overflow* (BOF), ou estouro de memória. Um *buffer* é uma área definida na memória do processo com o propósito de armazenar informações necessárias ao programa. O termo *definida* refere-se ao fato dessa área da memória possuir início e fim previamente estabelecidos. Nos exemplos acima, observou-se o uso da função *malloc*, da biblioteca padrão da linguagem *C*. Essa função faz a requisição de determinada quantidade de *bytes* de memória ao sistema operacional, que por sua vez verifica se há ou não a disponibilidade do recurso. Em caso de sucesso, é retornado o endereço do primeiro *byte* da área de memória cedida, como explicado anteriormente, localizada no *heap* do processo. A Figura 2 exemplifica um simples *buffer* de memória de sete *bytes*:

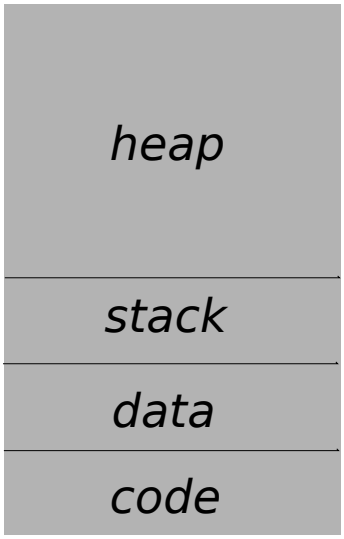

Figura 1 – Representação esquemática da organização da memória de um processo

Figura 2 – Representação esquemática de *buffer* de memória de sete *bytes*

O uso comum de um *buffer* envolve leitura e escrita em um ou mais *bytes* da região especificada. Para exemplificar o uso adequado de um *buffer*, assume-se que o retorno da função *malloc* é o endereço *0xFF01* e que foram alocados três *bytes* de memória. O Código 2.3 ilustra a manipulação de um *buffer* de memória alocado no *heap*.

Lista de códigos 2.3 Exemplo de *buffer overflow*

```
1 char * a = malloc(sizeof(char) * 3);
2 int i = 0;
3 a[i++] = 'a'; // i = 0: correto
4 a[i++] = 'b'; // i = 1: correto
5 a[i++] = 'c'; // i = 2: correto
6 char _a = a[i]; // i = 3: incorreto!
```

Na linha 1 é declarado um ponteiro para a área de memória em questão, *bytes* de *0xFF01* até *0xFF03*. Na linha 2 é declarado uma variável para ser usada como índice de acesso ao *buffer*. Na linha 3 tem-se a aritmética de ponteiro, onde soma-se o valor do endereço de memória do primeiro *byte* do ponteiro a (*0xFF01*) ao valor de i ($i = 0$), que resulta no endereço *0xFF01*. Calculado o endereço, é possível armazenar o caractere *'a'*, conforme mostrado na linha 3, o mesmo ocorre para as linhas 4 e 5. Na linha 6, porém, o endereço de memória calculado é *0xFF04* que extrapola o limite de a (*0xFF03*) ocasionando um BOF de leitura, onde se queria chegar.

Mais precisamente, o BOF do exemplo anterior se refere a *overflow* por que extrapolou-se o limite superior do *buffer*, isto é, acessou-se uma posição a mais que o *byte* mais significativo do *buffer*. Caso o endereço acessado fosse inferior ao *byte* menos significativo do *buffer*, este seria um *buffer underflow*. Ambos os casos de estouro de memória podem ocorrer tanto no *stack* quanto no *heap* e podem ocorrer tanto para leitura quanto para escrita de dados. As possibilidades apresentadas de ocorrer tal evento formam um total de oito tipos estouro de *buffer*. Entretanto, o nome comumente dado é somente *buffer overflow*, por ser mais comum em meio acadêmico e profissional. É nesse contexto que está fundamentado o conceito de BOF.

Conforme será apresentado na Subseção 2.4, dispõe-se de ferramentas SCA capazes de detectar faltas de *software*. Nesse sentido, a motivação complementar à apresentada no fim da Seção 2.2, é a de prover insumos para a identificação de pontos fracos em um determinado código. Utilizou-se o protótipo para extrair os insumos mínimos para a captura de *buffer overflows*, isto é, fez-se consulta de código de declarações de variáveis que sejam ponteiros.

2.4 Ferramentas relacionadas

Existem diversas ferramentas de análise estática de código, comerciais e livres (SPINROOT, 2014). Cada ferramenta possui características próprias que as diferenciam umas das outras. Na Tabela 2 é apresentada uma lista resumida de ferramentas existentes atualmente.

Ferramenta	**Link**
Analizo	www.analizo.org/
CodeSonar	www.grammatech.com/codesonar
Coverity	coverity.com
GoAnna	redlizards.com
HP Fortify	to.ly/zn1R
KlocWork	www.klocwork.com
LDRA	www.ldra.com/en
PMD	pmd.sourceforge.net
Semmle(Odasa)	semmle.com

Tabela 2 – Lista resumida de ferramentas de SCA

O *Analizo* funciona como extrator de métricas de qualidade, que podem ser usadas para diagnosticar o estado de um *software*. O *PMD* é uma ferramenta para análise de projetos em Java que diagnostica pontos do código que não seguem padrões de design. *CodeSonar*,*Coverity*, *GoAnna*, *HP Fortify*, *KlockWork* e *LDRA* são ferramentas de análise estática do código fonte com objetivo exclusivo de encontrar vulnerabilidades em códigos

em *C/C++* e *Java*. O *Semmle* é uma ferramenta que usa consulta de código e BI para trazer visualizações de projetos de software de uma forma mais exploratória (SPINROOT, 2014).

2.5 Consulta de código

Como será explicado na Subseção 3.1.2, uma das maneiras de se obter informações à cerca de um determinado código fonte é a partir da montagem de sua AST. Essa estrutura de dados possibilita a navegação entre seus nós em busca de dados relevantes à análise estática. A estrutura em árvore é ideal para obtenção de informações para cálculo da complexidade ciclomática de um código, como apresentado na Seção 2.2, porém não é tão indicada para outros tipos de extração de informação. Como exemplo, deseja-se identificar todas as funções que possuam um determinado número de linhas em seu corpo. Percebe-se que a estrutura de dados em árvore não é a mais indicada para esse cenário. A consulta de código surge para facilitar operações como a do exemplo descrito anteriormente e flexibilidade e recursos para outros tipos de análise.

A aplicação da consulta de código é bastante ampla na área de análise de *software*, análise arquitetural, engenharia reversa, verificação de consistência, verificação de padrões entre outros (HAGE et al., 2011). O uso e implementação dessas tecnologias segue o paradigma *extrair-abstrair-apresentar*, apresentado por Jurriaan Hage et. al.

Extrair extrair código fonte e mapeá-lo para alguma estrutura de armazenamento

Abstrair aplicar operações e consultas nessa estrutura para obter resultados

Apresentar apresentar os resultados

O presente trabalho se baseou no paradigma descrito acima. A extração será realizada através das ferramentas *Flex* e *Bison*; a abstração será realizada por meio de consultas às informações de código armazenadas em banco de dados e, por fim; a apresentação será, para fins de escopo, feita de forma textual.

A partir do levantamento bibliográfico, foi encontrado um número reduzido de ferramentas para consulta de código. Dentre esse conjunto, a maioria é proprietária e não possui versões para uso não-comercial. Nas próximas seções são apresentadas três ferramentas para consulta de código. A Subseção 2.5.1 exemplifica a consulta de código em sua forma básica, porém eficiente para determinados cenários. Nas Subseções 2.5.2 e 2.5.3 são apresentadas duas *Domain Specific Language(DSL)*, ou linguagens de domínio específico, para consulta de código.

2.5.1 Consultas textuais

As ferramentas de consulta de código dependem da complexidade do problema que se deseja solucionar. Para ocasiões simples, é possível fazer o uso de ferramentas nativas nos principais sistemas operacionais comuns, como o *GNU/Linux*. O comando *grep* é famoso por ser eficiente na busca por padrões de texto em arquivos. Para exemplificar o uso de consulta de código simples, é considerada a necessidade de se identificar todas referências de uma determinada função, *calc*, em um conjunto de arquivos. O comando do Código 2.4 pode ser interpretado da seguinte forma: *grep* faz a chamada ao programa de busca por padrões; o texto *calc(* especifica o padrão a ser buscado nos arquivos; o * em sistemas operacionais *GNU/Linux* tem o poder de referenciar todos arquivos e diretórios presentes na pasta atual; por último, a opção *-R* diz ao *grep* que, ao encontrar um diretório, deve-se aplicar o mesmo comando recursivamente até for possível.

Lista de códigos 2.4 Exemplo consulta de código

```
1 > grep calc( * -R
```

O Código 2.4 atende satisfatoriamente a necessidade especificada no exemplo, pois não há qualquer relacionamento entre outros elementos do código. Entretanto, a Engenharia de *Software* atual requere a resolução de problemas mais complexos, é vital que programadores mantenham seus códigos limpos e manuteníveis. Boas práticas de codificação atual se remetem ao número reduzido de linhas por função, ou método, para que determinado código apresente características boas de qualidade. É inviável, para um ser humano, a tarefa de contagem do número de linhas do código. Nesse sentido, a proposta da ferramenta *ezparser* foi visado para auxiliar o engenheiro de *software* em cenários em que há a necessidade de consulta de código.

2.5.2 *.QL*

Semmle[1], é uma empresa de métricas de *software* conhecida originalmente pelo seu produto, inicialmente chamado *SemmleCode*, um analisador de *software*. Pode ser usado para encontrar padrões de faltas de software, calcular métricas e para verificar padrões de design. O funcionamento da ferramenta se baseia em consulta de código através de uma linguagem de consulta orientada à objetos, a *.QL*[2] (lê-se dot-kiu-el), uma linguagem de consulta orientada à objetos para extração de informação de código fonte armazenado em banco de dados. É bastante similar à *SQL* e à programação orientada à objetos em Java. O ponto forte da *.QL* é a possibilidade de recursão para consultar informações hierárquicas,

[1] https://semmle.com/
[2] http://en.wikipedia.org/wiki/.QL

como classes, métodos e variáveis. No Código 2.5 é apresentado um trecho de consulta que obtém a relação de classes em Java com mais de dez métodos públicos em uma aplicação.

Lista de códigos 2.5 Exemplo de *.QL*

```
1 FROM Class c, int numofm
2 WHERE numofm = count(Method m | m.getDeclaringType() = c AND ↩
     m.hasModifier("public"))
3     AND numofm > 10
4 SELECT c.getPackage(), c, numofm
```

Na linha 1 são especificadas as entidades a serem usadas na consulta. *Class* é um tipo de dado que armazena todos os nomes de classes de determinado projeto. *int numofm* é uma variável inteira que será utilizada para armazenar o número de métodos públicos de cada classe. A facilidade da linguagem encontra-se na associação do método e da a classe ($m.getDeclaringType() = c$) e na restrição de que o método deve ser público ($m.hasModifier("public")$). Por fim, a consulta retornará uma listagem com as informações do nome do pacote da classe, o nome da classe e o número de métodos públicos. Entretanto, a ferramenta e sintaxe *.QL* são proprietárias, não sendo possível visualizar sua documentação.

2.5.3 *LINQ*

O *Language-Integrated Query* (lê-se *link*) é um conjunto de ferramentas introduzidas no Visual Studio 2008 que herda fortes características de consulta para a linguagem *C#* e *Visual Basic*. *LINQ* é um novo padrão, fácil de ser aprendido, para consultar e atualizar dados, e a tecnologia pode ser estendida para qualquer tipo de armazenamento de dados. O *Visual Studio* possui suporte ao *LINQ* para uso nas plataformas *.NET* da *Microsoft*. No Código 2.6 é apresentado um exemplo, utilizado para selecionar métodos em uma aplicação que sejam públicos e tenham mais de 30 linhas de código.

Lista de códigos 2.6 Exemplo de *LINQ*

```
1 FROM m in Application.Methods
2 WHERE m.NbLinesOfCode > 30 AND m.IsPublic
3 SELECT m
```

O *LINQ* possui traços semelhantes à *.QL*, apresentado na Subseção 2.5.2, se diferencia basicamente nos nomes das entidades, atributos e métodos de acesso. Por exemplo, na *.QL* a maneira para verificar se um método possui visibilidade pública, bastou-se especificar *m.hasModifier("public")*, enquanto na sintaxe *LINQ* o padrão é o atributo

m.IsPublic. Durante o processo de pesquisa, foram encontradas outras linguagens com sintaxe e propósito bastante semelhantes ao *LINQ*, a *Code Query Language CQL* e *CQ-Linq*[3].

A busca textual apresentada na Subseção 2.5.1 atende a necessidade de consulta até o momento em que não é necessário relacionamento entre outras partes do código. Nota-se a diferença de consulta proporcionado pelas ferramentas apresentadas nas Subseções 2.5.2 e 2.5.3. Essas duas dão ao usuário o poder de associar diferentes partes do código. Ambas as ferramentas para consulta de código são pagas e funcionam para as linguagens *Java* e *C#*, respectivamente. É nesse sentido optou-se pelo interesse de prototipação de uma ferramenta livre para consulta de códigos escritos em linguagem *C*. A Seção 2.6 aborda em mais detalhes o intuito da proposta deste trabalho.

2.6 Proposta

Uma vez que o leitor esteja familiarizado com os conceitos de análise estática e consulta de código, é possível apresentar a proposta do presente trabalho. Pretende-se desenvolver um protótipo de uma ferramenta de consulta de código. Dado um determinado código fonte escrito na linguagem de programação *C*, foram-se consultadas declarações de variáveis que tenham o tipo ponteiro e também declarações de funções. Na Figura 3 é apresentado o mapa mental/conceitual em alto nível do escopo do protótipo deste trabalho.

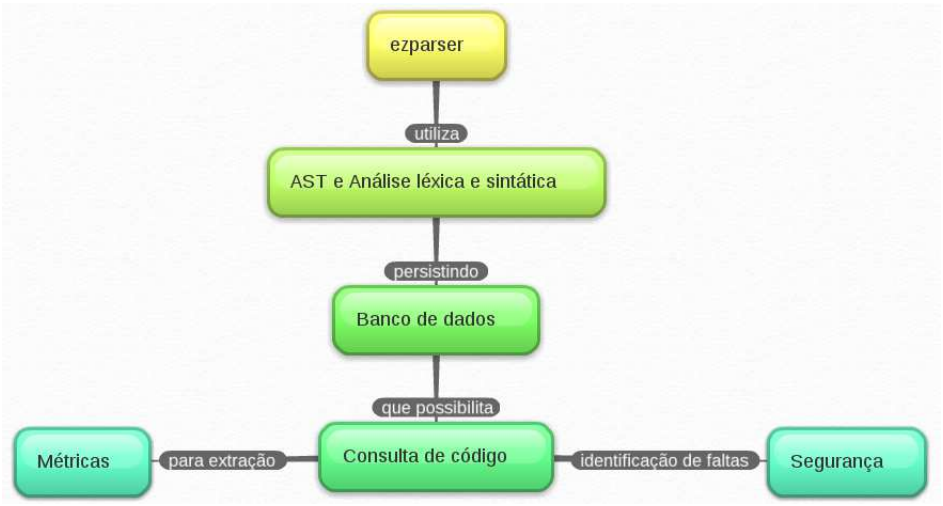

Figura 3 – Mapa conceitual do *ezparser*

O nome de ***ezparser***, do inglês *ez* é a forma sonora da palavra *easy* (lê-se ísi) que significa 'fácil', e *parser* vêm de uma das fases da análise sintática. Por esse motivo, os demais módulos do protótipo tem o prefixo *ez*, útil na distinção das estruturas desenvol-

[3] http://www.ndepend.com/Doc_CQLinq_CQL.aspx

vidas para a ferramenta. O intuito é de passar ao usuário a sensação de facilidade ao se analisar código fonte.

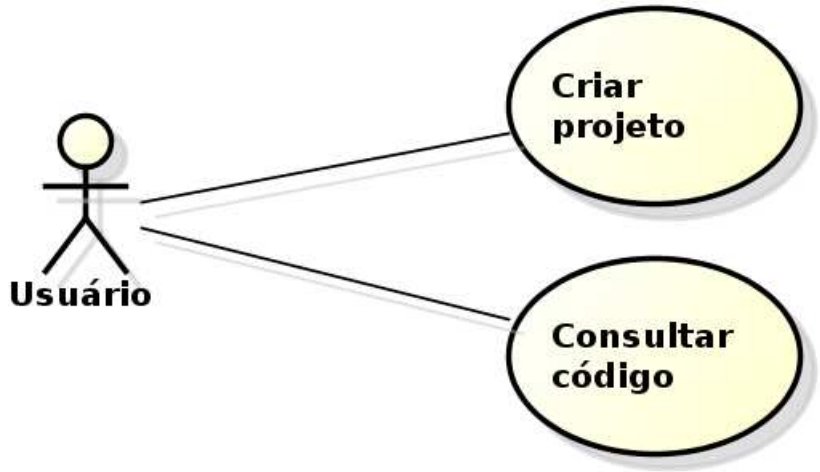

Figura 4 – Diagrama de casos de uso *ezparser*

Entende-se por *usuário* como qualquer interessado em consultar informações de um determinado código fonte. Para o contexto atual, foram visados desenvolvedores, porém, é possível que, outras ferramentas façam o uso de consulta de código. Como o principal do trabalho é somente a consulta, não houveram muitos casos de uso. Na Figura 4 é apresentado o modelo de casos de uso identificado para o protótipo.

Criar projeto visa manter a separação dos códigos no banco de dados. Conforme será apresentado na Seção 4.5, é necessário informar ao *ezparser* o nome do projeto. Caso o projeto já exista no banco de dados, o usuário será levado para um terminal onde poderá realizar consultas de códigos do projeto. Caso o projeto não exista, será criado um novo e os arquivos serão analisados e persistidos no banco.

Consultar código uma vez que o código fonte tenha sido analisado, o usuário pode realizar consultas de modo semelhante a um banco de dados comum, entretanto, usa-se somente sentenças de consultas em SQL.

3 Técnicas e ferramentas

Para gerar uma base de conhecimento, que fornecesse a fundamentação técnica necessária ao leitor, este capítulo é dedicado somente à apresentação de técnicas e ferramentas conhecidas que foram escolhidas para o desenvolvimento deste trabalho. Durante o decorrer do capítulo, serão destacadas as tarefas que tais técnicas e tecnologias desempenham no protótipo. Na Seção 3.1 são introduzidos os conceitos de AST e análise léxica e sintática. Posteriormente, na Seção 3.2, as tecnologias de linguagem de programação e de banco de dados são brevemente abordados. Por fim, na Seção 3.3, as ferramentas para gestão de configuração são discutidas em mais detalhe.

3.1 Técnicas de construção e análise

Nesta seção são apresentadas em detalhe as técnicas utilizadas para representação e análise de código. Inicia-se com a definição de árvore de sintaxe abstrata na Subseção 3.1.1. Posteriormente são apresentados conceitos base da SCA, a análise léxica e sintática de código. Esses conceitos têm implementações em diversas ferramentas, mas, para o contexto do trabalho, foram implementados pelas ferramentas *Flex* e *Bison*, ferramentas apresentadas na Subseção 3.1.2. Tendo-se definido o conceito de gramática no atual contexto, a Subseção 3.1.3 define a gramática utilizada como base para o desenvolvimento deste trabalho.

3.1.1 Árvore de sintaxe abstrata

Em ciência da computação, uma *abstract syntax tree* (AST), ou árvore de sintaxe abstrata, é a representação abstrata em árvore de um código fonte escrito em alguma linguagem de programação. Cada nó da árvore denota um elemento presente no código. A sintaxe é abstrata por não apresentar os detalhes específicos da linguagem real. Por exemplo, agrupamento de parêntesis são implícitos em uma estrutura de árvore e uma estrutura condicional *if-else* pode ser representada como um nó contendo três ramificações, um para a condição booleana e outros dois para caso verdadeiro ou falso (HOWE, 1985). A Figura 5 ilustra um exemplo de uma árvore de sintaxe abstrata, para uma simples declaração de variável seguida de uma atribuição.

3.1.2 Análise léxica e sintática

Flex e *Bison* são ferramentas construídas para criadores de compiladores e interpretadores, entretanto, ambas as ferramentas são bastante úteis em diversas aplicações

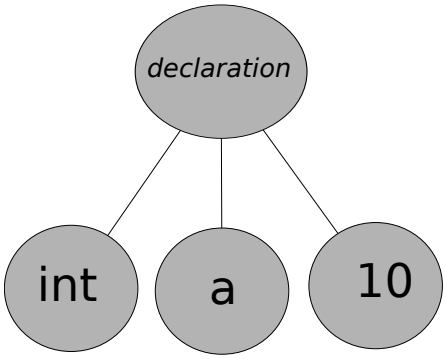

Figura 5 – Representação em AST de uma declaração *int a = 10;*

que interessem não-desenvolvedores de compiladores. Quaisquer aplicações que busquem por padrões em suas entradas ou que tenham sintaxe específica para entradas ou comandos são candidatas para usar *Flex* e *Bison*. Essas permitem rápida prototipação, fácil modificação e simples manutenção de programas (LEVINE, 2009). Por esse motivo, as duas ferramentas foram escolhidas para análise léxica e sintática dos códigos a serem analisados neste trabalho.

Os primeiros compiladores datam de 1950 e foram criados utilizando-se técnicas simples para analisar a sintaxe dos códigos fontes dos programas escritos na época. Durante os anos 60 esse campo de estudo ganhou atenção da academia e em meados de 1970 a análise sintática tornou-se um termo difundido (LEVINE, 2009). A experiência obtida possibilitou a abstração do trabalho em duas partes: análise léxica e análise sintática (ou *parsing*).

A análise léxica divide o texto de entrada em pedaços (ou *tokens*) com algum significado e o *parsing* trata da relação que os *tokens* capturados possuem uns com os outros. Considere o trecho de Código 3.1 na linguagem *C* como exemplo.

Lista de códigos 3.1 Exemplo de atribuição

```
1 a = b + c;
```

Nesse exemplo de código a análise léxica divide o trecho nos *tokens* *a*, *sinal de igual*, *b*, *sinal de soma*, *c* e *ponto-e-vírgula*. Posteriormente, o *parser* estipula que $b + c$ é uma expressão e que ela é atribuída à *a*.

O núcleo da ferramenta a ser desenvolvida neste trabalho, o *ezparser*, é composto por tais operações léxicas e sintáticas em nível maior de complexidade. Para se chegar ao nível de complexidade desejado para ferramenta é necessário antes a demonstração do potencial das ferramentas *Flex* e *Bison*. No Código 3.2 se descreve o comportamento o programa *word count*, simples ferramenta para reportar o número de linhas, palavras e caracteres encontrados em um arquivo. O código foi escrito na sintaxe específica do *Flex*,

que por sua vez um gera arquivo em *C* para que então seja possível gerar o programa executável.

Lista de códigos 3.2 Gramática léxica do *word count* (*wc.l*)

```
1  %{
2      int chars = 0;
3      int words = 0;
4      int lines = 0;
5  %}
6
7  %%
8
9  [a-zA-Z]+ { words++; chars += strlen(yytext); }
10 \n { chars++; lines++; }
11 . { chars++; }
12
13 %%
14 main(int argc, char **argv)
15 {
16     yylex();
17     printf("%8d%8d%8d\n", lines, words, chars);
18 }
```

A sintaxe do programa é familiar para quem programa em *C*. São três seções separadas por linhas contendo %%. A primeira seção contém declarações. A segunda seção contém uma lista de padrões de expressões regulares e ações. A terceira seção é formada por um trecho de código em *C* que é copiado para o programa gerado pelo *Flex*, nessa seção é comum haver pequenas rotinas relacionadas às ações dos padrões definidos na segunda seção.

A seção de declaração, localizada entre % e %, é copiada para o início do código em *C* gerado. No exemplo acima apenas são criadas variáveis para armazenar o número de linhas, palavras e caracteres. Na segunda seção cada padrão deve começar do início da linha sendo seguido por um trecho de código *C* a ser executado quando o padrão aparecer no texto de entrada. O código em *C* pode ser apenas um comando terminado por ponto-e-vírgula ou múltiplos comandos contidos em bloco de chaves.

Na segunda seção existem apenas três padrões de expressão regular. O primeiro *[a-zA-Z]+* significa qualquer palavra com um ou mais letras maiúsculas ou minúsculas, o sinal de + representa um modificador de número. A ação para este padrão incrementa o número de palavras e de caracteres capturados. O *Flex* usa a variável *yytext* para guardar

o texto capturado pela expressão regular. O segundo padrão, $\backslash n$, apenas faz referência à quebra de linha, logo a ação consiste basicamente em incrementar o número de linhas e de caracteres. O último padrão é um ponto, que, na sintaxe de expressão regular, é um caractere curinga. A ação é de atualizar o número de caracteres. Por fim, a última seção do arquivo *wc.l* é um código em *C* que chama a função *yylex()*, nome que a ferramenta *Flex* definiu para iniciar a análise léxica, e imprime os resultados. Para executar o programa é necessário que o *Flex* esteja instalado na máquina, após instalação, a sequência de comandos para construir o *wc* está descrita no Código 3.3.

Lista de códigos 3.3 Execução do *word count*

```
1 > flex wc.l
2 > gcc lex.yy.c -lfl
3 > ./a.out
4 >  The boy stood on the burning deck
5   shelling peanuts by the peck^D
6 > 2 12 63
```

O primeiro comando acima utiliza o *Flex* para transformar o *wc.l* para o arquivo auto-gerado *lex.yy.c*, que por sua vez pode ser compilado utilizando-se um compilador *C* que conheça a biblioteca *libfl.so*, utilizada pelo *Flex*. O resultado final é um programa com comportamento semelhante igual ao *wc* encontrado em sistemas operacionais *GNU/Linux*. A entrada do programa consiste em um texto qualquer que será analisado e apresentado o seu número de linhas, palavras e colunas.

Percebe-se que o exemplo anterior carece de técnica definida para fazer o uso dos *tokens*. Sendo assim, nos trechos código a seguir serão apresentados usos mais complexos do analisador léxico em conjunto com o analisador sintático *Bison*. Será implementada uma calculadora bastante simples capaz somente de somar, subtrair, multiplicar e dividir.

Lista de códigos 3.4 Exemplo de gramática léxica de uma calculadora (*calc.l*)

```
1  /* recognize tokens for the calculator and print them out */
2  %{
3      enum yytokentype {
4          NUMBER = 258,
5          ADD = 259,
6          SUB = 260,
7          MUL = 261,
8          DIV = 262,
9          ABS = 263,
10         EOL = 264
```

```
11      };
12      int yylval;
13  %}
14
15  %%
16  "+" { return ADD; }
17  "-" { return SUB; }
18  "*" { return MUL; }
19  "/" { return DIV; }
20  "|" { return ABS; }
21  [0-9]+      { yylval = atoi(yytext); return NUMBER; }
22  \n          { return EOL; }
23  [ \t]   { /* ignore whitespace */ }
24  .       { printf("Mystery character %c\n", *yytext); }
25  %%
26  main(int argc, char **argv)
27  {
28      int tok;
29      while(tok = yylex()) {
30          printf("%d", tok);
31          if(tok == NUMBER)
32              printf(" = %d\n", yylval);
33          else
34              printf("\n");
35      }
36  }
```

O Código 3.4 apresenta a descrição léxica que define *tokens* para uma calculadora simples. A diferença entre o exemplo anterior é a presença da expressão regular para capturar números *[0-9]+*. É possível repetir o procedimento de compilação como visto no Código 3.3, atentando-se para o nome do arquivo, e obter um programa que, dado algum trecho de texto, imprime apenas números inteiros e os sinais das quatro operações básicas da matemática.

Uma vez definida a gramática léxica da calculadora, pode-se voltar as atenções à gramática sintática, semelhante à léxica em termos de detecção de padrões, exceto que são *tokens* inteiros ao invés de caracteres. Assim como foi definido um arquivo para a análise léxica, será definido a gramática do Código 3.6 para definição da sintaxe da calculadora.

Para criar um analisador sintático (*parser*), é preciso alguma maneira formal para descrever as regras que o mesmo utiliza para transformar um sequência de *tokens* em uma AST. O tipo mais comum de linguagem usado pelos *parsers* é a *context-free grammar* (CFG), ou gramática livre de contexto. A forma padrão para escrever uma CFG é a

Backus-Naur Form (BNF) , criada com o nome de dois membros do comitê *Algol 60* por volta de 1960. Felizmente, BNF é bastante simples de se entender e a seguir encontra-se um exemplo para manusear expressões matemáticas como exemplo *1 * 2 + 3 * 4 + 5*.

Lista de códigos 3.5 Exemplo de sintaxe BNF (*bnf.y*)

```
1 <exp> ::= <factor>
2         | <exp> + <factor>
3 <factor> ::= NUMBER
4         | <factor> * NUMBER
```

No Código 3.5, cada linha é uma regra que diz respeito à criação de um nó da AST. Em BNF, '*::=*' pode ser lido como 'é um' e '/' como 'ou' outra maneira de criar um novo nó da AST do mesmo tipo. O nome à esquerda da regra é um *símbolo* ou *terminal*. Por convenção, todos os *tokens* são considerados símbolos. A Figura 6 ilustra a AST montada a partir da BNF do Código 3.5.

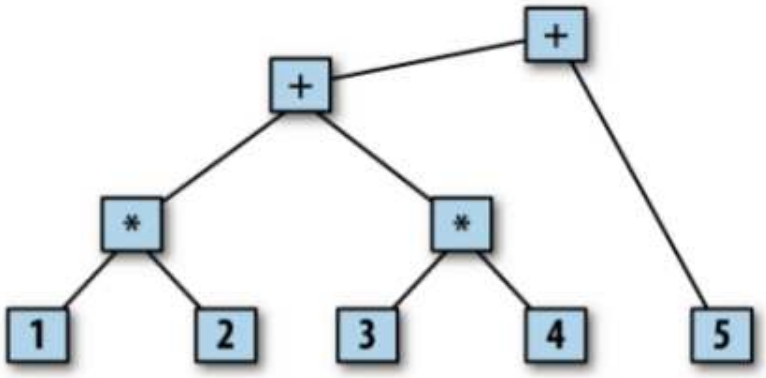

Figura 6 – AST gerada a partir da BNF do Código 3.5

Programas em *Bison* possuem mesmas características de programas em *Flex*, com estrutura de três partes e geração do código fonte em *C* a partir da gramática definida. As declarações de *tokens* do *Bison* também podem ser definidas como *%token*. Os símbolos que não forem declarados como *tokens* devem aparecer no lado esquerdo de uma regra da gramática. A segunda seção contém as regras no formato simplificado do BNF. O *Bison* usa um '*:*' no lugar de '*::=*' e '*;*' para marcar o fim de uma regra. Novamente, assim como o *Flex*, as ações podem vir dentro de blocos no final de cada regra.

Lista de códigos 3.6 Exemplo de gramática sintática de uma calculadora (*calc.y*)

```
1 /* simplest version of calculator */
2 %{
3     #include <stdio.h>
```

```
4  %}
5
6  /* declare tokens */
7  %token NUMBER
8  %token ADD SUB MUL DIV ABS
9  %token EOL
10 %%
11
12 calclist
13     : /* nothing */
14     | calclist exp EOL { printf("= %d\n", $1); }
15     ;
16
17 exp
18     : factor
19     | exp ADD factor { $$ = $1 + $3; }
20     | exp SUB factor { $$ = $1 - $3; }
21     ;
22
23 factor
24     : term
25     | factor MUL term { $$ = $1 * $3; }
26     | factor DIV term { $$ = $1 / $3; }
27     ;
28
29 term
30     : NUMBER
31     | ABS term { $$ = $2 >= 0? $2 : - $2; }
32     ;
33 %%
34 main(int argc, char **argv)
35 {
36     yyparse();
37 }
38 yyerror(char *s)
39 {
40     fprintf(stderr, "error: %s\n", s);
41 }
```

O *parsing* é feito automaticamente e as ações para cada uma das regras devem levar em consideração o contexto associado com o símbolo da regra. São criados também estruturas de dados básicas para a execução mínima de um programa em *Bison*, fazendo

com que cada símbolo nas regras possuam um valor. O valor de um símbolo do lado esquerdo do '*:*' é referenciado como '*$$*' no código da ação, essa notação será utilizada na Seção 4.3. Os valores do lado direito do '*:*' são referenciados como '*$1*', '*$2*' e assim por diante até o número de símbolos para aquela regra. Os valores dos *tokens* são os que estiverem em *yylval* quando o analisador léxico retornou o *token*.

Os valores dos outros símbolos da gramática são definidos pelo *parser*, a ser implementado pelas ações das regras. Na gramática do Código 3.6 os valores de *factor*, *term* e *exp* são os valores das expressões matemáticas que eles representam. As duas primeiras regras definem o símbolo *calcset* e implementam uma recursão que lê uma expressão terminada por uma quebra de linha, imprimindo o valor resultante. A definição do símbolo *calclist* usa a recursão novamente para implementar uma lista. A primeira regra é vazia e a segunda adiciona um item para uma lista. A ação da segunda regra imprime o valor do símbolo *exp* através da referência '*$2*'. O restante das regras implementam a calculadora da seguinte forma: as regras com operadores como '*exp ADD factor*' e '*ABS term*' fazem a aritmética com os valores encontrados nos símbolos, e assim por diante.

Por fim, são necessários passos extras para o funcionamento mínimo da calculadora do exemplo. É preciso informar à gramática do *Flex* as definições geradas pelo *Bison*. Na seção de definições do Código 3.4 deve-se incluir o arquivo *calc.tab.h*. A outra alteração se remete à remoção da função *main*, uma vez que a principal função a ser chamada encontra-se no *parser* gerado pelo *Bison*. Os comandos do Código 3.7 são necessários para gerar o programa da calculadora e realizar algumas operações matemáticas simples.

Lista de códigos 3.7 Execução da calculadora (*calc.sh*)

```
1  > bison -d calc.y
2  > flex calc.l
3  > gcc lex.yy.c calc.tab.c -o calc -lfl
4  > ./calc
5    2 + 3 * 4
6    = 14
7    2 * 3 + 4
8    = 10
9    20 / 4 - 2
10   = 3
11   20 - 4 / 2
12   = 18
```

3.1.3 Gramática *ANSI C*

Um insumo fundamental para este projeto é o entendimento da sintaxe da linguagem *C*. Como mencionado na Seção 2.5 é possível extrair informações de código através de diferentes técnicas, entretanto, a técnica selecionada para este projeto foi a realização de análise sintática através da ferramentas *Flex* e *Bison*, apresentadas na Subseção 3.1.2, por apresentarem características de maior tempo de existência e, consequentemente, estabilidade. Nesse sentido, procurou-se utilizar as gramáticas léxica e sintática padrões do *ANSI C*, uma vez que essas são utilizadas desde os primeiros compiladores da linguagem. Em 1985, Jeff Lee publicou a gramática do *Yacc* baseada na versão padrão do *ANSI C*, com suporte à especificação *Lex* (LEE, 2011). Essa gramática assume que a fase de tradução tenha sido finalizada, isto é, as diretivas de pré-processamento (comandos iniciados com #). As gramáticas podem ser encontradas nos anexos deste trabalho, no Anexo A.1 está a gramática léxica, interpretada pelo *Flex*, e no Anexo B.1 está a gramática sintática, interpretada pelo *Bison*.

3.2 Linguagens de programação e banco de dados

É comum em aplicações de *software* atualmente utilizarem múltiplias linguagens de programação, pois cada uma possui suas vantagens e desvantagens para cada cenário. A ferramenta *ezparser* abrange o uso de 3 linguagens populares. Na Subseção 3.2.1 será apresentada, brevemente, a linguagem *C* e seu papel neste trabalho. Nas Subseções 3.2.2 e 3.2.3 são apresentados o *C++* e o *PHP*, respectivamente, duas linguagens poderosas com propósitos divergentes.

3.2.1 Linguagem *C*

C é uma linguagem de programação de propósito geral, inicialmente desenvolvida por Dennis Ritchie entre 1969 e 1973. Assim como demais linguagens imperativas, o *C* possui suporte à programação estruturada, permitindo a criação de variáveis com diferentes escopos e ainda recursão de funções. Seu *design* foi pensado para que o código fonte seja traduzido eficientemente para linguagem de máquina, e por isso é utilizada consistentemente por aplicações que tiveram sua implementação em linguagem *assembly*, como o sistema operacional *Unix*. *C* é uma das linguagens de programação mais utilizadas que já existiram, consequentemente existem compiladores para a maioria das arquiteturas de computador e de sistemas operacionais disponíveis (RITCHIE et al., 1988).

C também apresenta a característica de ser fortemente *tipado*, isso significa que variáveis assumem somente tipo de dado durante toda a execução do programa. É também conhecida pela simplicidade nas operações, que traz facilidade ao traduzir programas em

C para *assembly*. A última característica é um dos fatores influenciadores para a escolha desta linguagem para a padrão a ser analisada pelo *ezparser*.

3.2.2 Linguagem *C++*

C++ é um linguagem de programação de propósito geral, visto como a evolução da linguagem *C* adicionada à orientação à objetos. É particularmente utilizada para aplicações com restrições de recursos de máquina, como sistemas embarcados e *softwares* de infra-estrutura. *C++* traz benefícios ao programador que investe tempo para dominar a linguagem, escrevendo código de qualidade. É uma linguagem para os que levam programação a sério. Existem bilhões de linhas de código em *C++* e isso acontece devido à promessa de compatibilidade com versões legadas da linguagem (STROUSTRUP, 2013).

Para o presente trabalho, deseja-se obter um *software* que seja mantido durante vários anos. Nesse sentido, definiu-se a linguagem de programação *C++* como principal para o desenvolvimento dos códigos fonte. Em conjunto, será utilizada a ferramenta *Make*, introduzida na Subseção 3.3.2, para compilação.

3.2.3 Linguagem *PHP*

PHP é uma linguagem de programação criada para manipulação de páginas para aplicações *web*, também é utilizada para propósito geral, em especial criação de *scripts*. A primeira versão do *PHP* foi chamada de *Personal Home Page Tools*, quando foi lançada por Rasmus Lerdorf em 1995 (LERDORF, 1995). Em janeiro de 2013 foi registrada a instalação do *PHP* em mais de 240 milhões de sites (IDE, 2013). O *PHP* também é conhecido pela sua documentação completa, disponível no próprio site do grupo que mantém a linguagem, *php.net*.

O suporte à expressões regulares do *PHP* foi o principal fator influenciador da escolha da linguagem para este trabalho. Na Seção 4.3 fica evidenciado o uso de *scripts* para a extração de regras da gramática sintática da linguagem *C*. Recursos como manipulação de *strings* e outros recursos de linguagens de alto nível do *PHP* foram necessários durante o desenvolvimento do *ezparser*, porém, escolhidos à ponto de não prejudicar a performance da ferramenta.

3.2.4 Banco de dados

Um banco de dados é um conjunto de dados. DBMSs, ou Sistemas gerenciadores de banco de dados, são aplicações de *software* que, como o próprio nome explica, gerencia um banco de dados, e que interagem com usuários, outras aplicações e o próprio banco de dados para analisar e capturar dados. São fundamentais em aplicações de nichos variados

e desempanam um papel importante para organização e abstração do tratamento de dados da aplicação de *software* (HELLERSTEIN et al., 2007).

PostgreSQL é um DBMS orientado à objetos desenvolvido de várias formas desde 1977. Começou com um projeto chamado *Ingres*, na Universidade de *Berkley* na Califórnia, EUA.

O *PostgreSQL* oferece o suporte à orientação à objetos e à esquemas. Esquemas são subdivisões de bancos de dados, aumentando o nível de granularidade e organização dos dados. O DBMS provém também suporte aos tipos customizados de variáveis além dos tipos nativos da ferramenta, assim como em uma linguagem de programação que permite ao programador criar suas próprias classes (WORSLEY et al., 2002). Nesse sentido, a ferramenta *PostgreSQL* foi selecionada para o desenvolvimento do atual projeto.

3.2.5 Linguagem *SQL*

SQL é uma sigla em inglês que significa Linguagem Estruturada de Consultas (*Structure Query Language*) e teve o seu surgimento nos laboratórios da IBM visando atender ao banco de dados *System R* (ELMASRI; NAVATHE, 2009). O SQL é uma linguagem para banco de dados que se tornou popular entre os diversos SGBD, ele foi amplamente adotada devido aos seus amplo suporte para definição de dados, consultas e atualizações. Contudo o esforço feito juntamente com a ANSI (American National Standards Institute) e a ISO (International Standards Organization) deram ao SQL enorme aceitação no mercado, fazendo com que várias empresas adaptassem tal linguagem para as suas necessidades especificas. Como exemplo de customização, pode-se citar o MySQL que fez diversas alterações em tal linguagem maior performance. Se por um lado a customização traz vantagens, por outro significa redução da portabilidade (MILANI, 2006).

A SQL nesse projeto será, inicialmente, a linguagem utilizada para fazer consultas no banco de dados.

3.3 Gerência de configuração

O presente trabalho apresenta o uso de práticas e ferramentas da Engenharia de *Software*. Na Subseção 3.3.1 será apresentada a ferramenta *git* e algumas métricas extraídas durante o desenvolvimento do *ezparser*. Posteriormente, na Subseção 3.3.2 será introduzida a ferramenta-base para a compilação dos códigos da ferramenta.

3.3.1 Controle de versão

Git é uma ferramenta de controle de versões distribuído, livre e de código aberto, criado para manusear projetos com velocidade e eficiência (TORVALDS, 2005). É uma

ferramenta de fácil aprendizado, apresentado rápida performance. Se compara com outras ferramentas de mesmo propósito como *Subversion* e *Control Version System*. O *Git* permite a criação distribuída de repositórios de código, logo, durante o desenvolvimento do presente trabalho, foi utilizado, além do repositório local, a plataforma para repositórios remotos, *Github*. Selecionou-se a ferramenta com o intuito de se manter versões dos códigos e dos textos deste trabalho.

3.3.2 Construção

Durante o desenvolvimento de um *software* escrito em linguagens compiladas, há a necessidade constante de se gerar o programa executável. O *Make* é um utilitário, capaz de construir programas e bibliotecas automaticamente a partir do código fonte. Para tal, é necessário que haja um arquivo, chamado *makefile* no diretório em que estão localizados os código fonte do projeto. Esse arquivo descreve como a compilação e construção deve ser seguida (FELDMAN, 1977).

Escolheu-se a ferramenta para se otimizar o tempo de desenvolvimento e organização do código. O *Make* é essencial para tarefas como a apresentada na Subseção 3.3.3, onde será apresentado o conceito de biblioteca estática, utilizada para auxiliar programas com vários número de elementos compiláveis.

3.3.3 Bibliotecas estáticas

Bibliotecas estáticas são uma simples coleção de arquivos-objeto (código fonte compilado, antes do executável), comprimidos em um único arquivo, convencionalmente terminado com extensão ".a". A vantagem do uso dessas bibliotecas encontra-se no ganho de tempo de compilação, não havendo a necessidade de recompilar o código inteiro repetidamente (WHEELER, 2003).

Complementarmente ao que será apresentado na Seção 4.2, os códigos fontes de cada um dos módulos do *ezparser* está contido em sua biblioteca estática própria. O programa final referencia os arquivos ".a" para gerar o programa executável, ao invés de recompilar módulo-a-módulo.

4 Ezparser

A proposta da ferramenta *ezparser* contempla nove módulos distintos: *ezdb, ezeval, ezast, ezdsl, ezast, ezrule, ezcparser, ezwatchdog* e *ezutil*. Cada uma das partes desempenham funções específicas. Em resumo, (i) o *ezdb* trata das iterações com o banco de dados; (ii) o *ezast* é responsável por representar uma *AST* para armazenar a estrutura do código fonte em estrutura de dados interna da ferramenta; (iii) o *ezeval* interpreta os dados do módulo *ezast*; (iv) o *ezdsl* se encarrega de converter a linguagem de consulta utilizada para *SQL*; (v) o *ezrule* é o maior módulo de todos, pois se encarrega de associar as regras da gramática da linguagem *C* com a estrutura de dados definida para este trabalho; (vi) o *ezcparser* se define como a integração dos módulos *ezrule*, *ezast* e *ezeval*; (vii) o *ezwatchdog* tem como única responsabilidade monitorar o estado dos arquivos a serem analisados pela ferramenta ezparser; por último, mas não menos importante, (viii) o *ezutil* forma um conjunto de classes se suporte, como gerenciamento de *logs*, arquivos de configuração, erros entre outros. Os módulos do protótipo foram pensados para se comparar aos de sistemas *Unix*, em que há a troca de serviços entre eles.

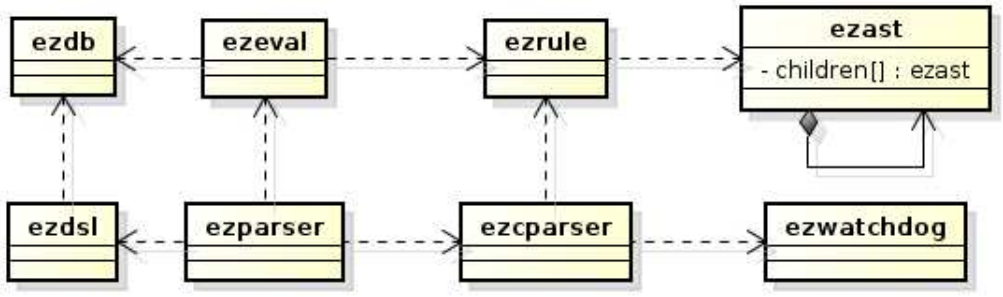

Figura 7 – Módulos *ezparser*

ezdb tem a responsabilidade de criar, atualizar e deletar esquemas e tabelas do banco de dados. Aqui são executadas as consultas vindas do módulo *ezdsl*. No atual estágio de desenvolvimento do projeto esse módulo interage somente com a ferramenta *psql*, ferramenta para executar e visualizar ações em bancos de dados *Postgres*.

ezeval é um dos módulos mais importante do *ezparser*. É responsável por receber o nó raiz do tipo *ezast* e fazer sua desmontagem de acordo com o tipo de nó encontrado nas ramificações da árvore. Durante a desmontagem AST, para cada expressão que tenha sentido completo, como exemplo uma atribuição, um condicional, é invocado o módulo *ezdb* para salvar aquela expressão no banco de dados.

ezast como explicado na Subseção 3.1.1, esse módulo é a implementação de uma AST. Um dos principais motivos para a escolha da linguagem *C++* para o projeto é o fato de haver suporte à programação orientação à objetos. Esse paradigma auxilia neste

módulo pois é necessário definir diversas estruturas de dados que possuam estruturas comuns. O *ezast* é o tipo mais genérico de árvore existente no *ezparser*, ao longo do desenvolvimento foram especializadas ASTs, estruturas de dados específicas para seu tipo de nó, como exemplo uma estrutura de repetição *for*, deve armazenar 4 informações diferentes: a inicialização, a condicional, o bloco de código e a pós-execução (a ser invocado após cada iteração do bloco).

ezdsl: este é o módulo responsável por traduzir linguagem de consulta de código para *SQL*. O estágio atual do projeto ainda **não** implementa essa funcionalidade pois não considerou-se um atrativo para a primeira versão do *ezparser*. Entretanto é interessante deixar claro o objetivo da existência do presente módulo para que no futuro seja possível processar consultas como vistas nas ferramentas apresentadas no Seção 2.5.

ezrule é o módulo mais volumoso atualmente. Nele são encontradas as ações das regras *Bison* para gramática da linguagem *C*. As ações são invocadas para cada vez que é encontrado um padrão, como apresentado na Subseção 3.1.2, e ao decorrer da execução do *parsing*, o *ezrule* cuida da montagem da AST do programa. Deve haver bastante dependência entre os módulos *ezrule* e *ezast*, pois é necessário se conhecer os tipos disponíveis de AST no *ezparser*.

ezcparser este módulo atua como integrador dos módulos *ezrule*, *ezeval* e *ezast*. É responsável pelo gerenciamento da montagem e desmontagem da AST.

ezwatchdog o nome do inglês significa *cão vigia*, neste contexto, vigiando os arquivos a serem passados para o *ezparser*. Note que esse módulo, assim como o *ezdsl* **não** está implementado. A principal tarefa do *ezwatchdog* é informar ao *parser* que determinados arquivos foram modificados para então analisá-los novamente e alterar, no banco de dados, os trechos afetados por tais mudanças. A complexidade dessa funcionalidade ainda é muito alta para ser implementada em tempo hábil para apresentação deste trabalho.

ezutil é o conjunto de auxiliares, como *ezlog*, *ezini* e *ezerror*. O *ezlog* cria uma classe estática com quatro níveis diferentes de *log* de sistema, para maior flexibilidade durante a apresentação de informações do *ezparser*, como exemplo, algumas informações só devem ser exibidas quando o programa estiver em modo *debug*. O *ezini* trata de ler configurações em um arquivo de inicialização com extensão '*.ini*'. Por fim, o *ezerror* tem a responsabilidade de manusear todas as mensagens de erro ao longo dos módulos do *ezparser*.

4.1 Metodologia de desenvolvimento

O *ezparser* está em seus estágios iniciais de desenvolvimento para este trabalho. Se percebido seu potencial de auxílio à engenheiros de *software*, a ferramenta pode apresentar

interesse em desenvolvimento colaborativo. Visando esse contexto, foi trabalhado um fluxo de desenvolvimento das funcionalidades básicas, as regras. A Figura 8 ilustra 7 etapas, definidas durante a construção inicial do *ezparser*.

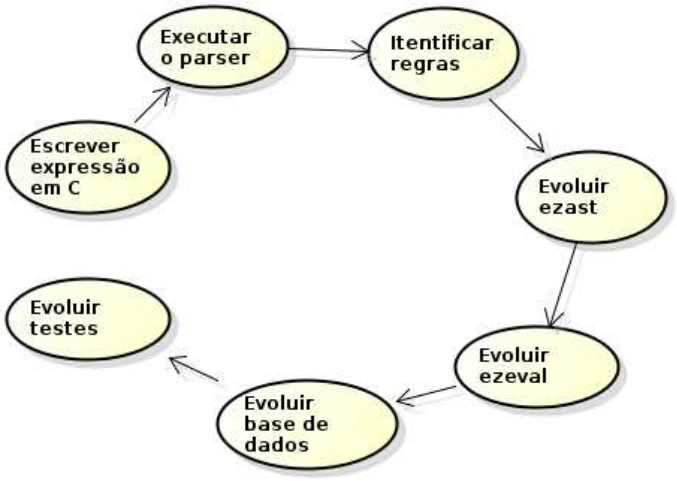

Figura 8 – Ciclo de desenvolvimento para o *ezparser*

Escrever expressão em *C* consiste em submeter um determinado trecho de código em *C* ao *ezparser*. Para exemplificar, considere uma simples atribuição 'int myvar = 2;'.

Executar o *parser* a saída de depuração do *ezparser* mostra todas as regras utilizadas para se analisar o trecho de código. O Código 4.1 mostra como é a saída. Note que é necessário se acessar várias regras para se realizar uma simples atribuição.

Identificar regras a partir da saída textual, é possível descobrir qual método da classe *ezrule* está responsável por tratar a regra. O nome do método é o nome da saída da linha apresentada no Código 4.1.

Evoluir *ezast* é necessário criar um tipo de *ezast* para cada tipo de regra. Como exemplo, é necessário criar uma estrutura de dados diferente para uma condicional e outra para uma estrutura de iteração. Infelizmente não houve tempo hábil para implementar outros tipos de *ezast*.

Evoluir *ezeval* à medida que houverem mais tipos de *ezast*, devem haver também modos para interpretá-las, isto é, a fase de desmontagem da *ezast* deve ser específica pelo seu tipo.

Evoluir base de dados sempre que necessário, deve-se evoluir o banco de dados, criando-se mais tabelas, sempre atento para não gerar conflitos com estruturas anteriores. Um ponto principal dessa tarefa é a consciência de que deve-se criar tabelas de modo que, ao ser utilizada pelo usuário da ferramenta, seja fácil consultar informações do código.

Evoluir testes para quase todos os módulos existe um diretório chamado 'tests',

destinado para criação de novos testes à medida do possível. No estado atual do *ezparser*, não foi utilizada biblioteca para gerenciar testes, ou seja, os mesmo são feitos manualmente com objetivo de verificar se os elementos dos módulos funcionam como o esperado. É um ponto que deve ser melhorado no futuro.

No intuito de contemplar um número maior de desenvolvedores para a ferramente, decidiu-se deixá-la livre (*GNU/GPL*), ou seja, os códigos estão abertos no repositório *git*, conforme mostrado no Apêndice A.

Lista de códigos 4.1 Atribuição int c

```
1  * Calling type_specifier__INT
2  * Calling declaration_specifiers__type_specifier
3  * Calling direct_declarator__IDENTIFIER
4  IDENTIFIER ['myvar'] is at line 1, column 9 and scope 0
5  * Calling declarator__direct_declarator
6  Calling constant__I_CONSTANT
7  2.000000
8  Calling primary_expression__constant
9  Calling postfix_expression__primary_expression
10 Calling unary_expression__postfix_expression
11 Calling cast_expression__unary_expression
12 Calling multiplicative_expression__cast_expression
13 Calling additive_expression__multiplicative_expression
14 Calling shift_expression__additive_expression
15 Calling relational_expression__shift_expression
16 Calling equality_expression__relational_expression
17 Calling and_expression__equality_expression
18 Calling exclusive_or_expression__and_expression
19 Calling inclusive_or_expression__exclusive_or_expression
20 Calling logical_and_expression__inclusive_or_expression
21 Calling logical_or_expression__logical_and_expression
22 Calling conditional_expression__logical_or_expression
23 Calling assignment_expression__conditional_expression
24 Calling initializer__assignment_expression
25 * Calling init_declarator__declarator_EQ_initializer
26 * Calling init_declarator_list__init_declarator
27 * Calling ↩
      declaration__declaration_specifiers_init_declarator_list_SCOLON↩

28 * Calling external_declaration__declaration
29 * Calling translation_unit__external_declaration
```

Antes de se encerrar esta seção, observa-se no Código 4.2 o método principal do *ezparser*. Procurou-se seguir o desenvolvimento baseado na Figura 8, de modo que haja rápida aprendizagem.

Lista de códigos 4.2 Método principal do *ezparser*

```
1  int ezparser::run(int argc, char ** argv)
2  {
3      ezlog::debug("Running ezparser...");
4      if(ezopt::process_opt(argc, argv) != 0)
5      {
6          ezlog::log(ezlog::ERROR, "Could not run ezparser, command line arguments are wrong!");
7          return 1;
8      }
9
10     ezdb * db;
11     ezdsl * dsl;
12     ezeval * evaluator;
13     ezcparser * parser;
14     string cmd, * sql;
15
16
17     if(!ezparser::project_exist(ezopt::projectname))
18     {
19         parser = new ezcparser;
20         evaluator = new ezeval;
21
22         ezlog::info("New project, parsing all " + to_string(ezopt::file_list.size()) + " files");
23
24         // Parse one file at a time, so we don't suck all the memory out of the system
25         for(int i_file = 0; i_file < ezopt::file_list.size(); i_file++)
26         {
27             ezast * ast = parser->parse_file(ezopt::file_list[i_file]);
28             if(!ast)
29             {
30                 ezlog::log(ezlog::ERROR, "Something went wrong during the parsing phase, see detailed
```

```
                    messages above");
31                  return 1;
32              }
33
34              if(evaluator->eval(ast) != NULL)
35              {
36                  ezlog::log(ezlog::ERROR, "Something went wrong↩
                        during the evaluation phase, see detailed ↩
                        messages above");
37                  return 1;
38              }
39          }
40      }
41
42      show_help();
43
44      db = ezdb::get_instance();
45      dsl = new ezdsl;
46
47      cout << "ez> ";
48      while(getline(cin, cmd))
49      {
50          //cout << "ez> ";
51
52          if(cmd == "exit")
53              break;
54          if(cmd == "help")
55          {
56              show_help();
57              cout << "ez> ";
58              continue;
59          }
60
61          if(cmd.size() == 0)
62          {
63              cout << "ez> ";
64              continue;
65          }
66
67          sql = dsl->to_sql(cmd);
68          if(!sql)
69          {
```

```
70              ezlog::log(ezlog::ERROR, "Something went wrong ↩
                   while querying, see detailed messages above");
71              cout << "ez> ";
72              continue;
73          }
74
75          vector< vector<string> > result = db->query_sql(*sql);
76
77          for(int i = 0; i < result.size(); i++)
78          {
79              string tuple = "";
80              for(int j = 0; j < result[i].size(); j++)
81              {
82                  tuple += string("'") + result[i][j] + string("↩
                       ', ");
83              }
84              cout << tuple << endl;
85          }
86          cout << "ez> ";
87      }
88
89      cout << endl;
90
91      return 0;
92  }
```

No Código 4.2 está a principal função do *ezparser*. Na linha 1 encontram-se os argumentos passados pelo usuário na linha de comando. A linha 4 faz o tratamento dos argumentos passados, verificando se há coerência entre os valores passados ao programa. Nas linhas 10 a 14 são declaradas algumas variáveis importantes para o funcionamento. Na linha 10 encontra-se uma referência para o *ezdb*, classe responsável por interações com o banco de dados. Na linha 11 encontra-se a instância da *ezdsl*, módulo responsável por tratar as consultas antes de serem enviadas ao banco. A linha 12 declara o *ezeval*, responsável por receber uma *ezast* e fazer o desmanche, inserindo as informações extraídas em banco de dados. Na linha 13 está a referência para o *ezcparser*, a classe responsável por fazer uso dos códigos gerados pelo *Flex* e *Bison*. A linha 25 contém o laço de iteração principal da fase de *parsing*, em que cada arquivo encontrado pelo *ezparser* é submetido à análise léxica e sintática, é gerada uma *ezast* para cada arquivo, como mostrado na linha 27. A linha 34 desempenha a função inversa à linha 27, pois aqui são percorridos todos os nós da árvore, persistindo-se as informações no banco de maneira controlada. Uma vez que toda a fase de *pasing* e persistência em banco é finalizada, o programa

segue para o seu laço principal, de interação com o usuário. A linha 48 recebe o comando informado pelo usuário, enquanto que nas linhas 52 à 65, são definidos fluxos básicos dependendo do comando requisitado. Tem-se na linha 67 o tratamento do comando para a SQL final. Nessa fase do *ezparser*, apenas comandos que sejam para consulta de dados foram permitidos. Finalmente, a SQL tratada é repassada à instância do *ezdb*, que submete a consulta para o banco de dados *PostgreSQL*. O formato dos resultados será retornado em um vetor de vetor de palavras, em que a primeira linha do vetor contém os nomes das colunas das tabelas e as linhas seguintes contém os respectivos valores. Cabe agora ao usuário a interpretação dos mesmos.

4.2 Organização do código

A Figura 9 ilustra a disposição dos elementos de código do *ezparser*. Na introdução deste capítulo, a responsabilidade de cada um dos módulos é discorrida em detalhes para que nesta seção seja explicado o modo de organização dos arquivos e a construção dos binários do projeto.

Figura 9 – Organização da raiz do *ezparser*

Os diretório *db*, *dsl*, *parser*, *util* e *watchdog* contém, respectivamente, os módulos *ezdb*, *ezdsl*, *ezcparser*, *ezutil* e *ezwatchdog*. Os demais módulos principais encontram-se na Figura 10, onde os diretórios *ast*, *eval* e *rules* contém, respectivamente *ezast*, *ezeval* e *ezrules*. Encontra-se um arquivo *makefile*, responsável por compilar todo o módulo, para cada módulo. O arquivo mais importante para a compilação do projeto é o da raiz da Figura 9, pois nele encontram-se as dependências de cada módulo. Tais dependências

estão na forma de biblioteca estática do *C++*, ou seja, arquivos com extensão '*.a*' que contém os códigos objeto dos módulos. Essas bibliotecas são fundamentais para o presente projeto, pois evita a necessidade de se compilar todo o código de uma única vez. Uma vez que sejam declaradas as bibliotecas como dependências, o *makefile* aponta os diretórios da localização das mesmas, cada um dos submódulos é compilado a partir da compilação do *ezparser*.

Figura 10 – Organização da raiz do *parser* de *C*

No estado atual de desenvolvimento há, aproximadamente, **9 mil linhas** de código, produzidas manualmente e automaticamente. Muitas linhas de código foram geradas a partir das gramáticas utilizadas pelo *Flex* e *Bison* (aproximadamente 5 mil linhas). O restante das milhares de linhas foram criadas através de *scripts PHP* e pouca parte de forma estritamente manual. Entende-se que, mesmo sendo geradas automaticamente, a ideia e arquitetura do código foi de total autoria do autor deste trabalho. Consequentemente, o tempo de compilação total do projeto é de cerca de **3 minutos**, um número considerável de códigos fonte.

4.3 Divisão das regras da gramática

Na Subseção 3.1.2 foi discutido o modo que o *Bison* executa ações para cada regra de *tokens* encontrados. Tem-se uma ação a ser invocada em *C* para cada regra da gramática. Havia um desafio complexo ao se definir um conjunto de ações para todas as regras existentes na gramática *ANSI C*, no Anexo B.1. Como definir um bloco de código para cada regra ?

Lista de códigos 4.3 Uso do *ezrule*

```
1 arquivo: gramatica.y
2  simbolo
3       : simbolo2 simbolo3 { = ezrule.simbolo_simbolo2_simbolo3↩
           (1,2); }
4       | simbolo4 simbolo5 { = ezrule.simbolo_simbolo4_simbolo5↩
           (1,2); }
5       ;
6
7 arquivo: ezrule.h
8  ezast * ezrule::simbolo_simbolo2_simbolo3(ezast * simbolo2, ↩
      ezast * simbolo3)
9  ezast * ezrule::simbolo_simbolo4_simbolo5(ezast * simbolo4, ↩
      ezast * simbolo5)
```

O Código 4.3 é a resposta. A criação de uma classe com a responsabilidade de gerenciar tais regras, a *ezrule*, foi a solução encontrada para o problema. Essa classe foi gerada automaticamente por *scripts* em *PHP*. Foi gerado um método para cada regra da gramática sintática, fazendo com que todas as regras sejam cobertas. O desafio então encontra-se em realizar uma força-tarefa para implementar cada uma das regras e construir a AST do código, assim como exemplificado na Subseção 3.1.2. Nesse contexto, foi criado um modelo de colaboração para a ferramenta *ezparser*, na Seção 4.1.

4.4 Cobertura das regras

A gramática da *ANSI C* contém 77 regras, cada regra contém várias sub regras. A combinação entre regra e sub regra totaliza 274 definições distintas capazes de analisar qualquer código fonte escrito na linguagem *C* (não inclui diretivas de pré-processamento de código).

Para identificar quais das regras já foram implementadas e quais ainda não foram, o autor do trabalho definiu um padrão simples. Para cada regra implementada, insere-se um '*' na mensagem de depuração. No Código 4.1 é possível perceber as regras que estão implementadas e as que não estão. Observe que o fato de haver mensagem de depuração não significa que a regra esteja implementada, a mensagem funciona apenas como um meio de auxiliar o desenvolvedor para evoluir a ferramenta.

4.5 Como usar

Esta seção visa ensinar ao leitor como operar o *ezparser*. A Figura 11 representa o fluxo lógico das informações processadas pelo protótipo, onde há divisão em etapas para

melhor entendimento do usuário. Há a descrição detalhada sobre cada uma das etapas e em seguida, no Código 4.4, será apresentado o seu uso na linha de comando.

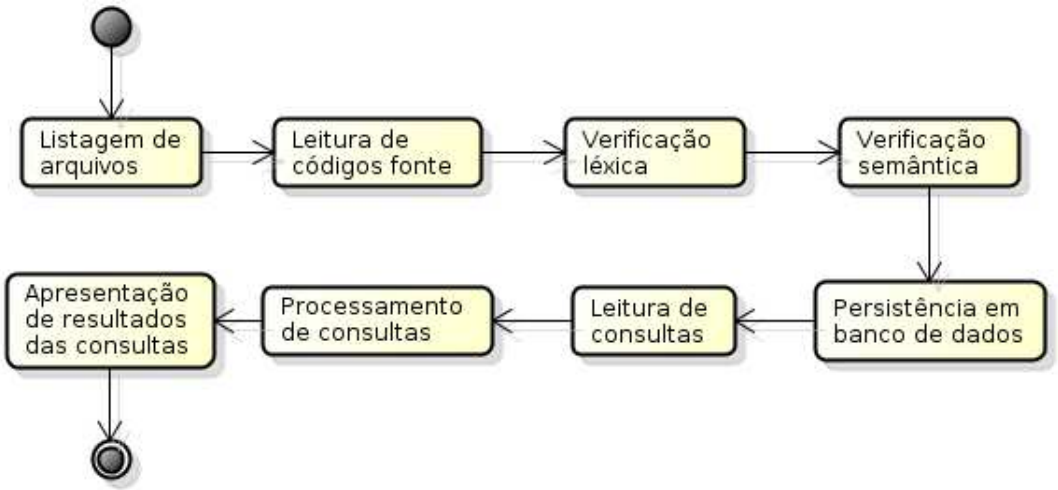

Figura 11 – Fluxo de execução do *ezparser*

Listagem de arquivos usuário informa ao *ezparser* qual(is) arquivos devem ser extraídas as informações.

Leitura de códigos fonte para cada arquivo, é coletado seu código fonte para as fases de análise léxica e sintática.

Verificação léxica é realizada a checagem léxica do código fonte dos arquivos, verificando-se caracteres inválidos.

Verificação sintática (semântica) após a análise léxica, tem-se a análise sintática, em que é verificado se o texto do código fonte é realmente escrito na sintaxe estabelecida pelo *ANSI C*.

Persistência em banco de dados é a próxima etapa, uma vez validados os códigos fontes, os mesmos são persistidos no banco de dados, conforme a estrutura a especificada pelo *ezparser*.

Leitura de consultas o usuário submete uma consulta (em SQL).

Processamento de consultas é a consulta em si. Nessa etapa são consultadas as informações dos códigos de acordo com a consulta submetida pelo usuário na etapa anterior

Apresentação de resultados das consultas é apresentado em forma textual a listagem contendo os resultados obtidos com a consulta.

Lista de códigos 4.4 Exemplo de uso do *ezparser*

```
1 $ ./ezparser -p my_project -f parser/tests/file_to_be_parsed.c
```

```
2 [INFO : 08:10:28:617.395] New project, parsing all 1 files
3 ezparser 0.0.1 - Code Query Tool
4 Currently there are not many commands:
5  - help: display this help message
6  - exit: quits the program
7  - <sql statements>: SQL used to directly query postgresql ↩
       data
8 [INFO : 08:10:28:658.214] Database connection OK
9 ez> SELECT * FROM identifiers
10 'identifier_id', 'name', 'line', '_column', 'scope', 'pointers↩
     ', 'file_id_files', 'type_id_types',
11 '1', 'c', '1', '5', '0', '0', '1', '1',
12 ez> exit
```

5 Conclusões

No sentido de interpretar os resultados de forma prática, é preciso o estabelecimento de um contexto. Será considerado um exemplo simples, em que se deseja conhecer os possíveis pontos vulneráveis à *buffer overflow*, apresentado na Subseção 2.3.1. Sabe-se que esse tipo de defeito ocorre sempre quando há o uso de ponteiros, pois certamente haverá o uso deste para acessar uma posição da memória. Logo deve-se ter o conhecimento de todas as variáveis do que sejam ponteiro ao longo do código. Essa necessidade requere um mecanismo de busca capaz de retornar a referência da expressão, nome de arquivo e número de linha, provendo ao usuário insumos para investigação.

No Código 5.1 tem-se um código bastante simples. Na linha 1 é declarada uma variável inteira. Na linha 2 é declarada uma variável como referência para um valor inteiro. Note que a diferença entre as duas linhas é somente o '*'.

Lista de códigos 5.1 Código fonte

```
1 int c;
2 int * my_pointer;
```

Lista de códigos 5.2 Consulta de ponteiro

```
1 chaws@worrior:~/UnB/TCC/tcc/src/ezparser(master *)$ ./ezparser↩
      -p my_project -f parser/tests/file_to_be_parsed.c
2 [INFO : 09:34:44:139.863] New project, parsing all 1 files
3 ezparser 0.0.1 - Code Query Tool
4 Currently there are not many commands:
5  - help: display this help message
6  - exit: quits the program
7  - <sql statements>: SQL used to directly query postgresql ↩
      data
8 [INFO : 09:34:44:171.801] Database connection OK
9 ez> SELECT * FROM identifiers
10 'identifier_id', 'name', 'line', '_column', 'scope', 'pointers↩
     ', 'file_id_files', 'type_id_types',
11 '1', 'c', '1', '5', '0', '0', '1', '1',
12 '4', 'my_pointer', '2', '16', '0', '1', '1', '1',
13 ez> SELECT * FROM identifiers WHERE pointers > 0
```

```
14 'identifier_id', 'name', 'line', '_column', 'scope', 'pointers↩
     ', 'file_id_files', 'type_id_types',
15 '4', 'my_pointer', '2', '16', '0', '1', '1', '1',
16 ez> exit
```

No Código 5.2, é apresentada a listagem obtida através da consulta do Código 5.1, que permite ao usuário conhecer todas as variáveis declaradas no programa. Na linha 9 são consultadas todas as variáveis, enquanto na linha 13 é estabelecido o critério 'pointers > 0'. Este critério faz parte da regra estabelecida pelo *ezparser*, em que todas as variáveis possuem o atributo 'pointers', que significa o número de ponteiros. Essa abordagem foi usada porquê a linguagem de programação *C* permite ao programador criar ponteiros múltiplos, cabendo tratamentos devidos durante o código. Observe que os campos 'name', 'line', '_column', 'file_id_files' e 'type_id_types' são triviais. Já o campo 'scope' merece mais atenção. Esse campo se refere ao escopo em que a variável foi declarada, e será incrementado toda vez que houver uma abertura de chaves '{' e decrementada sempre que houver fechamento '}'.

No estado atual deste trabalho, 20% das regras da gramática sintática foram implementadas, representando um percentual moderadamente médio. Tal resultado se deve ao fato da complexidade em manter o volume de código gerado. Apesar de apenas menos da metade das regras estarem implementadas, consideram-se os resultados bastante positivos, pois em um curto período de tempo foi possível se construir uma ferramenta que faz o *parsing* de um código e o insere em um banco de dados.

A vantagem do protótipo *ezparser* é a automatização de buscas específicas. Atualmente, os usuários desse tipo de aplicação são os próprios desenvolvedores do código analisado. A necessidade de encontrar pontos semelhantes ao contexto proposto no início do capítulo é recorrente e geralmente realizada de forma manual. Conclui-se que ferramentas para consulta de código, fornecem ao usuário a flexibilidade de se obter tais informações específicas, retirando-se a necessidade de modificação do módulo de extração de código (fase *Extrair* apresentada na introdução da Seção 2.5).

Observou-se o uso intenso combinado de diversas disciplinas estudadas durante curso de engenharia de *software*. A construção do *ezparser* necessitou de habilidades com tecnologias *Bison*, *Flex*, e suas sintaxes; com as linguagens de programação *C*, *C++*, *PHP* e *SQL*; com ferramentas de controle de configuração e construção *git* e *Make*; com os paradigmas de programação orientado à objetos e estruturado; e com as boas práticas de programação, essa última visando deixar o código como um atrativo para novos contribuidores. Nesse sentido, acredita-se que o objetivo do trabalho de conclusão de curso foi atingido satisfatoriamente.

5.1 Trabalhos futuros

Conclui-se também que, a consulta de código possui um potencial ainda inexplorado. Durante a fundamentação teórica do presente trabalho, notou-se a ausência de relacionamento da consulta de código com ferramentas SCA. Sugere-se que a combinação entre ferramentas de análise estática e de consulta de código possa resultar em análises mais robustas e flexíveis. Como exemplo, parte-se do contexto do *Analizo*, uma ferramenta livre, para métricas de *software* contemplando múltiplas linguagens para análise e visualização de código (TERCEIRO, 2008).

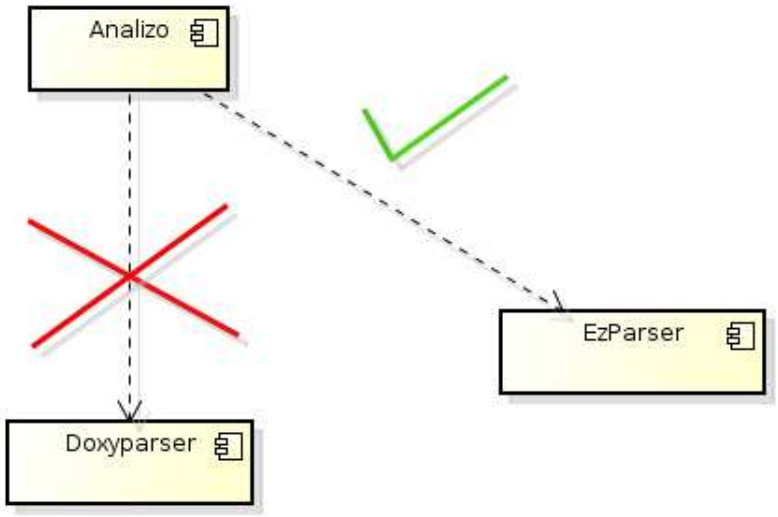

Figura 12 – Integração *Analizo* e *ezparser*

Essa ferramenta trabalha em conjunto com o *Doxyparse*[1], o *paser* do *Doxygen*[2]. Entretanto, criando-se um cenário futuro, havendo necessidade de aumentar o tipos de métricas extraídas pelo *Analizo*, haverá também a necessidade de evolução do *Doxyparse*. Baseando-se nos modestos resultados apresentados no Capítulo 4, o *ezparser*, em seu estágio avançado de desenvolvimento, pode ser considerado como alternativa ao *Doxyparse*. A Figura 12 ilustra, de forma simples, a posição da ferramenta em relação ao *Analizo*

[1] https://github.com/terceiro/doxyparse
[2] http://www.stack.nl/ dimitri/doxygen/

Referências

ANDREY, K. *Static code analysis*. 2012. Disponível em: <http://www.codeproject.com/Tips/344663/Static-code-analysis>. Acesso em: 2014-11-01. Citado na página 27.

BALZAROTTI, D. et al. Saner: Composing static and dynamic analysis to validate sanitization in web applications. p. 15, May 2008. Citado na página 23.

CHRISTEY, S. et al. *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. 2011. Disponível em: <http://cwe.mitre.org/top25/index.html#Listing>. Acesso em: 2014-11-01. Citado 2 vezes nas páginas 29 e 30.

CODEBASES. Codebases - millions of lines of code. 2013. Disponível em: <http://www.informationisbeautiful.net/visualizations/million-lines-of-code/>. Citado na página 23.

COWAN, C. et al. Buffer overflows: Attacks and defenses for the vulnerability of the decade. p. 11, 2000. Citado na página 31.

ELMASRI, R.; NAVATHE, S. *Sistemas de banco de dados*. [S.l.: s.n.], 2009. 149 p. Citado na página 49.

FELDMAN, S. *GNU Make*. 1977. Disponível em: <http://www.gnu.org/software/make/>. Acesso em: 2014-11-01. Citado na página 50.

HAGE, J. et al. A comparative study of code query technologies. p. 12, April 2011. ISSN 0924-3275. Citado 2 vezes nas páginas 24 e 34.

HELLERSTEIN, J. M. et al. Architecture of a database system. p. 119, 2007. Citado na página 49.

HOWE, D. *Free On-line Dictionary of Computing*. 1985. Disponível em: <http://foldoc.org/>. Acesso em: 2014-11-01. Citado na página 39.

IDE, A. *PHP just grows and grows*. 2013. Disponível em: <http://news.netcraft.com/archives/2013/01/31/php-just-grows-grows.html>. Acesso em: 2014-11-01. Citado na página 48.

LAPLANTE, P. A. *What every engineer should know about software engineering*. [S.l.]: Taylor and Francis Group, 2007. ISBN 978-0-8493-7228-5. Citado na página 23.

LEE, J. *ANSI C Yacc grammar*. 2011. Disponível em: <http://www.quut.com/c/ANSI-C-grammar-y.html>. Acesso em: 2014-11-01. Citado na página 47.

LERDORF, R. *PHP: Hypertext Preprocessor*. 1995. Disponível em: <http://php.net/>. Acesso em: 2014-11-01. Citado na página 48.

LEVINE, J. R. *Flex and Bison*. [S.l.]: O'Reilly Media, 2009. ISBN 978-0-596-15597-1. Citado na página 40.

MCCABE, T. J. A complexity measure. p. 308–320, 1976. Citado na página 29.

MEIRELLES, P. R. M. *Monitoramento de métricas de código-fonte em projetos de software livre*. Tese (Doutorado) — Universidade de São Paulo, Maio 2013. Citado na página 29.

MILANI, A. *MySQL guia do programador*. [S.l.: s.n.], 2006. 27 p. Citado na página 49.

PRESSMAN, R. S. *Software engineering: a practioner's approach*. 5th. ed. [S.l.]: Thomas Casson, 2001. ISBN 0073655783. Citado na página 23.

RITCHIE, D. et al. *C Programming Language*. [S.l.]: Prentice Hall, 1988. ISBN 9780133086218. Citado na página 47.

SPINROOT. Static source code analysis tools for c. 2014. Disponível em: <http://spinroot.com/static/>. Citado 2 vezes nas páginas 33 e 34.

STROUSTRUP, B. *The C++ Programming Language*. [S.l.]: Addison–Wesley, 2013. ISBN 0321563840. Citado na página 48.

TANENBAUM, A. S. *Modern Operating Systems*. [S.l.]: Prentice Hall, 2007. ISBN 978-0136006633. Citado na página 31.

TERCEIRO, A. *Analizo*. 2008. Disponível em: <http://www.analizo.org/>. Acesso em: 2014-11-01. Citado na página 65.

TORVALDS, L. *Git*. 2005. Disponível em: <http://git-scm.com/>. Acesso em: 2014-11-01. Citado na página 49.

WHEELER, D. A. Program library howto. p. 25, 2003. Citado na página 50.

WICHMANN, B. A. et al. Industrial perspective on static analysis. p. 7, March 1995. Citado na página 23.

WILLIAMS, J. *Dangerous Function*. 2009. Disponível em: <https://www.owasp.org/index.php/Dangerous_Function>. Acesso em: 2014-11-01. Citado na página 28.

WORSLEY, J. C. et al. *Practical PostgreSQL*. [S.l.]: O'Reilly and Associates, 2002. ISBN 1-56592-846-6. Citado na página 49.

Apêndices

APÊNDICE A – Instalação

Como apresentado na Seção 3.2, é necessário a instalação de algumas dependências na máquina para a correta execução deste protótipo. Primeiramente, definiu-se que o ambiente de desenvolvimento seria GNU/Linux. Posteriormente, definiu-se que seria utilizada a distribuição *Debian*. Para instalar as dependências, basta rodar o comando abaixo:

sudo apt-get install php5-cli flex bison postgresql-9.1 postgresql-server-dev-9.1 g++ git

Uma vez que todas as dependências tenham sido instaladas, é necessário que seja baixado o código fonte do *ezparser*, que se encontra na url <https://github.com/chaws/tcc.git>. Para baixar os códigos, digite o seguinte comando:

git clone https://github.com/chaws/tcc.git

O último comando faz uma cópia local do presente trabalho, contendo ambas parte escrita e código fonte. Para dar continuidade, navegue entre as pastas até encontrar 'src/ezparser'. Por fim, digite o seguinte comando para que o projeto seja configurado:

make

O tempo de compilação leva entorno de 3 minutos e uma vez finalizado é possível executar o protótipo assim como apresentado na Seção 4.5.

Anexos

ANEXO A – Gramática léxica do C-2011

Lista de códigos A.1 Gramática léxica do *ANSI C*

```
%e  1019
%p  2807
%n  371
%k  284
%a  1213
%o  1117

%option yylineno

O   [0-7]
D   [0-9]
NZ  [1-9]
L   [a-zA-Z_]
A   [a-zA-Z_0-9]
H   [a-fA-F0-9]
HP  (0[xX])
E   ([Ee][+-]?{D}+)
P   ([Pp][+-]?{D}+)
FS  (f|F|l|L)
IS  (((u|U)(l|L|ll|LL)?)|((l|L|ll|LL)(u|U)?))
CP  (u|U|L)
SP  (u8|u|U|L)
ES  (\\([\'"\?\\abfnrtv]|[0-7]{1,3}|x[a-fA-F0-9]+))
WS  [ \t\v\f]
NEWLINE [\n]

%{
#include <cstdio>
#include <cstdlib>
#include <string.h>
#include <iostream>
#include <ezast.h>
#include <c_grammar_2011.tab.h>

#define YY_DECL extern "C" int yylex()
#define sym_type(identifier) IDENTIFIER /* with no symbol table, fake it */
extern void yyerror(const char *);  /* prints grammar violation message */
static void comment(void);
static int check_type(void);

// Keep track of lines & columns
extern int __current_line;
extern int __current_column;
extern int __current_scope;

%}

%%
"/*"                    { comment(); }
"//".*                  { /* consume //-comment */ __current_column = 0; }
```

```
51
52 "auto"                { __current_column += strlen("auto");                return(AUTO);↩
       }
53 "break"               { __current_column += strlen("break");               return(BREAK)↩
       ; }
54 "case"                { __current_column += strlen("case");                return(CASE);↩
       }
55 "char"                { __current_column += strlen("char");                return(CHAR);↩
       }
56 "const"               { __current_column += strlen("const");               return(CONST)↩
       ; }
57 "continue"            { __current_column += strlen("continue");            return(↩
       CONTINUE); }
58 "default"             { __current_column += strlen("default");             return(↩
       DEFAULT); }
59 "do"                  { __current_column += strlen("do");                  return(DO); }
60 "double"              { __current_column += strlen("double");              return(DOUBLE↩
       ); }
61 "else"                { __current_column += strlen("else");                return(ELSE);↩
       }
62 "enum"                { __current_column += strlen("enum");                return(ENUM);↩
       }
63 "extern"              { __current_column += strlen("extern");              return(EXTERN↩
       ); }
64 "float"               { __current_column += strlen("float");               return(FLOAT)↩
       ; }
65 "for"                 { __current_column += strlen("for");                 return(FOR); ↩
       }
66 "goto"                { __current_column += strlen("goto");                return(GOTO);↩
       }
67 "if"                  { __current_column += strlen("if");                  return(IF); }
68 "inline"              { __current_column += strlen("inline");              return(INLINE↩
       ); }
69 "int"                 { __current_column += strlen("int");                 return(INT); ↩
       }
70 "long"                { __current_column += strlen("long");                return(LONG);↩
       }
71 "register"            { __current_column += strlen("register");            return(↩
       REGISTER); }
72 "restrict"            { __current_column += strlen("restrict");            return(↩
       RESTRICT); }
73 "return"              { __current_column += strlen("return");              return(RETURN↩
       ); }
74 "short"               { __current_column += strlen("short");               return(SHORT)↩
       ; }
75 "signed"              { __current_column += strlen("signed");              return(SIGNED↩
       ); }
76 "sizeof"              { __current_column += strlen("sizeof");              return(SIZEOF↩
       ); }
77 "static"              { __current_column += strlen("static");              return(STATIC↩
       ); }
78 "struct"              { __current_column += strlen("struct");              return(STRUCT↩
       ); }
79 "switch"              { __current_column += strlen("switch");              return(SWITCH↩
       ); }
80 "typedef"             { __current_column += strlen("typedef");             return(↩
       TYPEDEF); }
81 "union"               { __current_column += strlen("union");               return(UNION)↩
       ; }
82 "unsigned"            { __current_column += strlen("unsigned");            return(↩
```

```
    UNSIGNED); }
"void"                  { __current_column += strlen("void");                return(VOID);
    }
"volatile"              { __current_column += strlen("volatile");            return(
    VOLATILE); }
"while"                 { __current_column += strlen("while");               return(WHILE)
    ; }
"_Alignas"              { __current_column += strlen("_Alignas");            return
    ALIGNAS; }
"_Alignof"              { __current_column += strlen("_Alignof");            return
    ALIGNOF; }
"_Atomic"               { __current_column += strlen("_Atomic");             return ATOMIC
    ; }
"_Bool"                 { __current_column += strlen("_Bool");               return BOOL;
    }
"_Complex"              { __current_column += strlen("_Complex");            return
    COMPLEX; }
"_Generic"              { __current_column += strlen("_Generic");            return
    GENERIC; }
"_Imaginary"            { __current_column += strlen("_Imaginary");          return
    IMAGINARY; }
"_Noreturn"             { __current_column += strlen("_Noreturn");           return
    NORETURN; }
"_Static_assert"        { __current_column += strlen("_Static_assert");      return
    STATIC_ASSERT; }
"_Thread_local"         { __current_column += strlen("_Thread_local");       return
    THREAD_LOCAL; }
"__func__"              { __current_column += strlen("__func__");            return
    FUNC_NAME; }

{L}{A}*                     { yylval.word = strdup(yytext); __current_column +=
    strlen(yytext); return check_type(); }
{HP}{H}+{IS}?               { yylval.number = atof(yytext); __current_column +=
    strlen(yytext); return I_CONSTANT; }
{NZ}{D}*{IS}?               { yylval.number = atof(yytext); __current_column +=
    strlen(yytext); return I_CONSTANT; }
"0"{O}*{IS}?                { yylval.number = atof(yytext); __current_column +=
    strlen(yytext); return I_CONSTANT; }
{CP}?"'"([^'\\\n]|{ES})+"'" { yylval.number = atof(yytext); __current_column +=
    strlen(yytext); return I_CONSTANT; }
{D}+{E}{FS}?                { yylval.number = atof(yytext); __current_column +=
    strlen(yytext); return F_CONSTANT; }
{D}*"."{D}+{E}?{FS}?        { yylval.number = atof(yytext); __current_column +=
    strlen(yytext); return F_CONSTANT; }
{D}+"."{E}?{FS}?            { yylval.number = atof(yytext); __current_column +=
    strlen(yytext); return F_CONSTANT; }
{HP}{H}+{P}{FS}?            { yylval.number = atof(yytext); __current_column +=
    strlen(yytext); return F_CONSTANT; }
{HP}{H}*"."{H}+{P}{FS}?     { yylval.number = atof(yytext); __current_column +=
    strlen(yytext); return F_CONSTANT; }
{HP}{H}+"."{P}{FS}?         { yylval.number = atof(yytext); __current_column +=
    strlen(yytext); return F_CONSTANT; }
({SP}?\"([^"\\\n]|{ES})*\"{WS}*)+   { yylval.word = strdup(yytext); __current_column
    += strlen(yytext);  return STRING_LITERAL; }

"..."                   { __current_column += strlen("..."); return ELLIPSIS; }
">>="                   { __current_column += strlen(">>="); return RIGHT_ASSIGN; }
"<<="                   { __current_column += strlen("<<="); return LEFT_ASSIGN; }
"+="                    { __current_column += strlen("+="); return ADD_ASSIGN; }
"-="                    { __current_column += strlen("-="); return SUB_ASSIGN; }
```

```
116 "*="                    { __current_column += strlen("*="); return MUL_ASSIGN; }
117 "/="                    { __current_column += strlen("/="); return DIV_ASSIGN; }
118 "%="                    { __current_column += strlen("%="); return MOD_ASSIGN; }
119 "&="                    { __current_column += strlen("&="); return AND_ASSIGN; }
120 "^="                    { __current_column += strlen("^="); return XOR_ASSIGN; }
121 "|="                    { __current_column += strlen("|="); return OR_ASSIGN; }
122 ">>"                    { __current_column += strlen(">>"); return RIGHT_OP; }
123 "<<"                    { __current_column += strlen("<<"); return LEFT_OP; }
124 "++"                    { __current_column += strlen("++"); return INC_OP; }
125 "--"                    { __current_column += strlen("--"); return DEC_OP; }
126 "->"                    { __current_column += strlen("->"); return PTR_OP; }
127 "&&"                    { __current_column += strlen("&&"); return AND_OP; }
128 "||"                    { __current_column += strlen("||"); return OR_OP; }
129 "<="                    { __current_column += strlen("<="); return LE_OP; }
130 ">="                    { __current_column += strlen(">="); return GE_OP; }
131 "=="                    { __current_column += strlen("=="); return EQ_OP; }
132 "!="                    { __current_column += strlen("!="); return NE_OP; }
133 ";"                     { __current_column += strlen(";"); return ';'; }
134 ("{"|"<%")              { __current_column += strlen("{"); __current_scope++; return ↩
        '{'; }
135 ("}"|"%>")              { __current_column += strlen("}"); __current_scope--; return ↩
        '}'; }
136 ","                     { __current_column += strlen(","); return ','; }
137 ":"                     { __current_column += strlen(":"); return ':'; }
138 "="                     { __current_column += strlen("="); return '='; }
139 "("                     { __current_column += strlen("("); return '('; }
140 ")"                     { __current_column += strlen(")"); return ')'; }
141 ("["|"<:")              { __current_column += strlen("["); return '['; }
142 ("]"|":>")              { __current_column += strlen("]"); return ']'; }
143 "."                     { __current_column += strlen("."); return '.'; }
144 "&"                     { __current_column += strlen("&"); return '&'; }
145 "!"                     { __current_column += strlen("!"); return '!'; }
146 "~"                     { __current_column += strlen("~"); return '~'; }
147 "-"                     { __current_column += strlen("-"); return '-'; }
148 "+"                     { __current_column += strlen("+"); return '+'; }
149 "*"                     { __current_column += strlen("*"); return '*'; }
150 "/"                     { __current_column += strlen("/"); return '/'; }
151 "%"                     { __current_column += strlen("%"); return '%'; }
152 "<"                     { __current_column += strlen("<"); return '<'; }
153 ">"                     { __current_column += strlen(">"); return '>'; }
154 "^"                     { __current_column += strlen("^"); return '^'; }
155 "|"                     { __current_column += strlen("|"); return '|'; }
156 "?"                     { __current_column += strlen("?"); return '?'; }
157
158 {WS}                    { __current_column++; /* whitespace separates tokens */ }
159 {NEWLINE}               { __current_line++; __current_column = 0; }
160 .                       { __current_column++; /* discard bad characters */ }
161
162 %%
163
164 int yywrap(void)        /* called at end of input */
165 {
166     return 1;           /* terminate now */
167 }
168
169 static void comment(void)
170 {
171     int c;
172
173     while ((c = yyinput()) != 0)
```

```
174         if (c == '*')
175         {
176             while ((c = yyinput()) == '*')
177                 ;
178 
179             if (c == '/')
180                 return;
181 
182             if (c == 0)
183                 break;
184         }
185         else if(c == '\n')
186         {
187             __current_line++;
188             __current_column = 0;
189         }
190     yyerror("unterminated comment");
191 }
192 
193 static int check_type(void)
194 {
195     switch (sym_type(yytext))
196     {
197     case TYPEDEF_NAME:                /* previously defined */
198         return TYPEDEF_NAME;
199     case ENUMERATION_CONSTANT:        /* previously defined */
200         return ENUMERATION_CONSTANT;
201     default:                          /* includes undefined */
202         return IDENTIFIER;
203     }
204 }
```

ANEXO B – Gramática sintática do C-2011

Lista de códigos B.1 Gramática sintática do *ANSI C*

```
%{
#include <cstdio>
#include <ezast.h>
#include <ezrule.h>
#include <c_grammar_2011.tab.h>

// stuff from flex that bison needs to know about
extern "C" int yylex();
extern "C" FILE *yyin;
extern ezrule rule;
void yyerror(const char *s);

%}

/* Declare data types to be used */
%union {
    double number;
    char * word;
    ezast * ast;
}

/* AUTO-GENERATED TYPES */
%type <ast> primary_expression
%type <ast> constant
%type <ast> enumeration_constant
%type <ast> string
%type <ast> generic_selection
%type <ast> generic_assoc_list
%type <ast> generic_association
%type <ast> postfix_expression
%type <ast> argument_expression_list
%type <ast> unary_expression
%type <ast> unary_operator
%type <ast> cast_expression
%type <ast> multiplicative_expression
%type <ast> additive_expression
%type <ast> shift_expression
%type <ast> relational_expression
%type <ast> equality_expression
%type <ast> and_expression
%type <ast> exclusive_or_expression
%type <ast> inclusive_or_expression
%type <ast> logical_and_expression
%type <ast> logical_or_expression
%type <ast> conditional_expression
%type <ast> assignment_expression
%type <ast> assignment_operator
%type <ast> expression
%type <ast> constant_expression
%type <ast> declaration
```

```
51 %type <ast> declaration_specifiers
52 %type <ast> init_declarator_list
53 %type <ast> init_declarator
54 %type <ast> storage_class_specifier
55 %type <ast> type_specifier
56 %type <ast> struct_or_union_specifier
57 %type <ast> struct_or_union
58 %type <ast> struct_declaration_list
59 %type <ast> struct_declaration
60 %type <ast> specifier_qualifier_list
61 %type <ast> struct_declarator_list
62 %type <ast> struct_declarator
63 %type <ast> enum_specifier
64 %type <ast> enumerator_list
65 %type <ast> enumerator
66 %type <ast> atomic_type_specifier
67 %type <ast> type_qualifier
68 %type <ast> function_specifier
69 %type <ast> alignment_specifier
70 %type <ast> declarator
71 %type <ast> direct_declarator
72 %type <ast> pointer
73 %type <ast> type_qualifier_list
74 %type <ast> parameter_type_list
75 %type <ast> parameter_list
76 %type <ast> parameter_declaration
77 %type <ast> identifier_list
78 %type <ast> type_name
79 %type <ast> abstract_declarator
80 %type <ast> direct_abstract_declarator
81 %type <ast> initializer
82 %type <ast> initializer_list
83 %type <ast> designation
84 %type <ast> designator_list
85 %type <ast> designator
86 %type <ast> static_assert_declaration
87 %type <ast> statement
88 %type <ast> labeled_statement
89 %type <ast> compound_statement
90 %type <ast> block_item_list
91 %type <ast> block_item
92 %type <ast> expression_statement
93 %type <ast> selection_statement
94 %type <ast> iteration_statement
95 %type <ast> jump_statement
96 %type <ast> translation_unit
97 %type <ast> external_declaration
98 %type <ast> function_definition
99 %type <ast> declaration_list
100
101 /* Manually added types */
102 %type <word> IDENTIFIER
103 %type <word> STRING_LITERAL
104 %type <word> TYPEDEF_NAME
105 %type <word> FUNC_NAME
106 %type <number> I_CONSTANT
107 %type <number> F_CONSTANT
108
109 %token  IDENTIFIER I_CONSTANT F_CONSTANT STRING_LITERAL FUNC_NAME SIZEOF
110 %token  PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
```

```
111 %token  AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
112 %token  SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
113 %token  XOR_ASSIGN OR_ASSIGN
114 %token  TYPEDEF_NAME ENUMERATION_CONSTANT
115 %token  TYPEDEF EXTERN STATIC AUTO REGISTER INLINE
116 %token  CONST RESTRICT VOLATILE
117 %token  BOOL CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE VOID
118 %token  COMPLEX IMAGINARY
119 %token  STRUCT UNION ENUM ELLIPSIS
120 %token  CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN
121 %token  ALIGNAS ALIGNOF ATOMIC GENERIC NORETURN STATIC_ASSERT THREAD_LOCAL
122
123 %start translation_unit
124
125 %locations
126 %%
127
128 primary_expression
129     : IDENTIFIER {$$ = rule.primary_expression__IDENTIFIER($1);}
130     | constant {$$ = rule.primary_expression__constant();}
131     | string {$$ = rule.primary_expression__string();}
132     | '(' expression ')' {$$ = rule.primary_expression__LP_expression_RP($2);}
133     | generic_selection {$$ = rule.primary_expression__generic_selection($1);}
134     ;
135
136 constant
137     : I_CONSTANT {$$ = rule.constant__I_CONSTANT($1);}
138     | F_CONSTANT {$$ = rule.constant__F_CONSTANT($1);}
139     | ENUMERATION_CONSTANT {$$ = rule.constant__ENUMERATION_CONSTANT();}
140     ;
141
142 enumeration_constant
143     : IDENTIFIER {$$ = rule.enumeration_constant__IDENTIFIER();}
144     ;
145
146 string
147     : STRING_LITERAL {$$ = rule.string__STRING_LITERAL($1);}
148     | FUNC_NAME {$$ = rule.string__FUNC_NAME($1);}
149     ;
150
151 generic_selection
152     : GENERIC '(' assignment_expression ',' generic_assoc_list ')' {$$ = rule.↩
            generic_selection__GENERIC_LP_assignment_expression_COMMA_generic_assoc_list_RP↩
            ($3, $5);}
153     ;
154
155 generic_assoc_list
156     : generic_association {$$ = rule.generic_assoc_list__generic_association($1);}
157     | generic_assoc_list ',' generic_association {$$ = rule.↩
            generic_assoc_list__generic_assoc_list_COMMA_generic_association($1, $3);}
158     ;
159
160 generic_association
161     : type_name ':' assignment_expression {$$ = rule.↩
            generic_association__type_name_COLON_assignment_expression($1, $3);}
162     | DEFAULT ':' assignment_expression {$$ = rule.↩
            generic_association__DEFAULT_COLON_assignment_expression($3);}
163     ;
164
165 postfix_expression
```

```
166     : primary_expression {$$ = rule.postfix_expression__primary_expression($1);}
167     | postfix_expression '[' expression ']' {$$ = rule.↩
            postfix_expression__postfix_expression_LSB_expression_RSB($1, $3);}
168     | postfix_expression '(' ')' {$$ = rule.↩
            postfix_expression__postfix_expression_LP_RP($1);}
169     | postfix_expression '(' argument_expression_list ')' {$$ = rule.↩
            postfix_expression__postfix_expression_LP_argument_expression_list_RP($1, $3)↩
            ;}
170     | postfix_expression '.' IDENTIFIER {$$ = rule.↩
            postfix_expression__postfix_expression_DOT_IDENTIFIER();}
171     | postfix_expression PTR_OP IDENTIFIER {$$ = rule.↩
            postfix_expression__postfix_expression_PTR_OP_IDENTIFIER();}
172     | postfix_expression INC_OP {$$ = rule.↩
            postfix_expression__postfix_expression_INC_OP();}
173     | postfix_expression DEC_OP {$$ = rule.↩
            postfix_expression__postfix_expression_DEC_OP();}
174     | '(' type_name ')' '{' initializer_list '}' {$$ = rule.↩
            postfix_expression__LP_type_name_RP_LCB_initializer_list_RCB();}
175     | '(' type_name ')' '{' initializer_list ',' '}' {$$ = rule.↩
            postfix_expression__LP_type_name_RP_LCB_initializer_list_COMMA_RCB();}
176     ;
177
178 argument_expression_list
179     : assignment_expression {$$ = rule.↩
            argument_expression_list__assignment_expression();}
180     | argument_expression_list ',' assignment_expression {$$ = rule.↩
            argument_expression_list__argument_expression_list_COMMA_assignment_expression↩
            ();}
181     ;
182
183 unary_expression
184     : postfix_expression {$$ = rule.unary_expression__postfix_expression();}
185     | INC_OP unary_expression {$$ = rule.unary_expression__INC_OP_unary_expression()↩
            ;}
186     | DEC_OP unary_expression {$$ = rule.unary_expression__DEC_OP_unary_expression()↩
            ;}
187     | unary_operator cast_expression {$$ = rule.↩
            unary_expression__unary_operator_cast_expression();}
188     | SIZEOF unary_expression {$$ = rule.unary_expression__SIZEOF_unary_expression()↩
            ;}
189     | SIZEOF '(' type_name ')' {$$ = rule.unary_expression__SIZEOF_LP_type_name_RP()↩
            ;}
190     | ALIGNOF '(' type_name ')' {$$ = rule.unary_expression__ALIGNOF_LP_type_name_RP↩
            ();}
191     ;
192
193 unary_operator
194     : '&' {$$ = rule.unary_operator__AMP();}
195     | '*' {$$ = rule.unary_operator__MULT();}
196     | '+' {$$ = rule.unary_operator__PLUS();}
197     | '-' {$$ = rule.unary_operator__MINUS();}
198     | '~' {$$ = rule.unary_operator__TILDE();}
199     | '!' {$$ = rule.unary_operator__EXM();}
200     ;
201
202 cast_expression
203     : unary_expression {$$ = rule.cast_expression__unary_expression();}
204     | '(' type_name ')' cast_expression {$$ = rule.↩
            cast_expression__LP_type_name_RP_cast_expression();}
205     ;
```

```
206
207 multiplicative_expression
208     : cast_expression {$$ = rule.multiplicative_expression__cast_expression();}
209     | multiplicative_expression '*' cast_expression {$$ = rule.↩
            multiplicative_expression__multiplicative_expression_MULT_cast_expression();}
210     | multiplicative_expression '/' cast_expression {$$ = rule.↩
            multiplicative_expression__multiplicative_expression_DIV_cast_expression();}
211     | multiplicative_expression '%' cast_expression {$$ = rule.↩
            multiplicative_expression__multiplicative_expression_MOD_cast_expression();}
212     ;
213
214 additive_expression
215     : multiplicative_expression {$$ = rule.↩
            additive_expression__multiplicative_expression();}
216     | additive_expression '+' multiplicative_expression {$$ = rule.↩
            additive_expression__additive_expression_PLUS_multiplicative_expression();}
217     | additive_expression '-' multiplicative_expression {$$ = rule.↩
            additive_expression__additive_expression_MINUS_multiplicative_expression();}
218     ;
219
220 shift_expression
221     : additive_expression {$$ = rule.shift_expression__additive_expression();}
222     | shift_expression LEFT_OP additive_expression {$$ = rule.↩
            shift_expression__shift_expression_LEFT_OP_additive_expression();}
223     | shift_expression RIGHT_OP additive_expression {$$ = rule.↩
            shift_expression__shift_expression_RIGHT_OP_additive_expression();}
224     ;
225
226 relational_expression
227     : shift_expression {$$ = rule.relational_expression__shift_expression();}
228     | relational_expression '<' shift_expression {$$ = rule.↩
            relational_expression__relational_expression_LT_shift_expression();}
229     | relational_expression '>' shift_expression {$$ = rule.↩
            relational_expression__relational_expression_GT_shift_expression();}
230     | relational_expression LE_OP shift_expression {$$ = rule.↩
            relational_expression__relational_expression_LE_OP_shift_expression();}
231     | relational_expression GE_OP shift_expression {$$ = rule.↩
            relational_expression__relational_expression_GE_OP_shift_expression();}
232     ;
233
234 equality_expression
235     : relational_expression {$$ = rule.equality_expression__relational_expression();}
236     | equality_expression EQ_OP relational_expression {$$ = rule.↩
            equality_expression__equality_expression_EQ_OP_relational_expression();}
237     | equality_expression NE_OP relational_expression {$$ = rule.↩
            equality_expression__equality_expression_NE_OP_relational_expression();}
238     ;
239
240 and_expression
241     : equality_expression {$$ = rule.and_expression__equality_expression();}
242     | and_expression '&' equality_expression {$$ = rule.↩
            and_expression__and_expression_AMP_equality_expression();}
243     ;
244
245 exclusive_or_expression
246     : and_expression {$$ = rule.exclusive_or_expression__and_expression();}
247     | exclusive_or_expression '^' and_expression {$$ = rule.↩
            exclusive_or_expression__exclusive_or_expression_XOR_and_expression();}
248     ;
249
```

```
250 inclusive_or_expression
251     : exclusive_or_expression {$$ = rule.↩
            inclusive_or_expression__exclusive_or_expression();}
252     | inclusive_or_expression '|' exclusive_or_expression {$$ = rule.↩
            inclusive_or_expression__inclusive_or_expression_OR_exclusive_or_expression()↩
            ;}
253     ;
254
255 logical_and_expression
256     : inclusive_or_expression {$$ = rule.↩
            logical_and_expression__inclusive_or_expression();}
257     | logical_and_expression AND_OP inclusive_or_expression {$$ = rule.↩
            logical_and_expression__logical_and_expression_AND_OP_inclusive_or_expression↩
            ();}
258     ;
259
260 logical_or_expression
261     : logical_and_expression {$$ = rule.logical_or_expression__logical_and_expression↩
            ();}
262     | logical_or_expression OR_OP logical_and_expression {$$ = rule.↩
            logical_or_expression__logical_or_expression_OR_OP_logical_and_expression();}
263     ;
264
265 conditional_expression
266     : logical_or_expression {$$ = rule.conditional_expression__logical_or_expression↩
            ();}
267     | logical_or_expression '?' expression ':' conditional_expression {$$ = rule.↩
            conditional_expression__logical_or_expression_QUM_expression_COLON_conditional_expression↩
            ();}
268     ;
269
270 assignment_expression
271     : conditional_expression {$$ = rule.assignment_expression__conditional_expression↩
            ();}
272     | unary_expression assignment_operator assignment_expression {$$ = rule.↩
            assignment_expression__unary_expression_assignment_operator_assignment_expression↩
            ();}
273     ;
274
275 assignment_operator
276     : '=' {$$ = rule.assignment_operator__EQ();}
277     | MUL_ASSIGN {$$ = rule.assignment_operator__MUL_ASSIGN();}
278     | DIV_ASSIGN {$$ = rule.assignment_operator__DIV_ASSIGN();}
279     | MOD_ASSIGN {$$ = rule.assignment_operator__MOD_ASSIGN();}
280     | ADD_ASSIGN {$$ = rule.assignment_operator__ADD_ASSIGN();}
281     | SUB_ASSIGN {$$ = rule.assignment_operator__SUB_ASSIGN();}
282     | LEFT_ASSIGN {$$ = rule.assignment_operator__LEFT_ASSIGN();}
283     | RIGHT_ASSIGN {$$ = rule.assignment_operator__RIGHT_ASSIGN();}
284     | AND_ASSIGN {$$ = rule.assignment_operator__AND_ASSIGN();}
285     | XOR_ASSIGN {$$ = rule.assignment_operator__XOR_ASSIGN();}
286     | OR_ASSIGN {$$ = rule.assignment_operator__OR_ASSIGN();}
287     ;
288
289 expression
290     : assignment_expression {$$ = rule.expression__assignment_expression();}
291     | expression ',' assignment_expression {$$ = rule.↩
            expression__expression_COMMA_assignment_expression();}
292     ;
293
294 constant_expression
```

```
295     : conditional_expression {$$ = rule.constant_expression__conditional_expression();}
296     ;
297
298 declaration
299     : declaration_specifiers ';' {$$ = rule.declaration__declaration_specifiers_SCOLON($1);}
300     | declaration_specifiers init_declarator_list ';' {$$ = rule.declaration__declaration_specifiers_init_declarator_list_SCOLON($1, $2);}
301     | static_assert_declaration {$$ = rule.declaration__static_assert_declaration($1);}
302     ;
303
304 declaration_specifiers
305     : storage_class_specifier declaration_specifiers {$$ = rule.declaration_specifiers__storage_class_specifier_declaration_specifiers();}
306     | storage_class_specifier {$$ = rule.declaration_specifiers__storage_class_specifier();}
307     | type_specifier declaration_specifiers {$$ = rule.declaration_specifiers__type_specifier_declaration_specifiers();}
308     | type_specifier {$$ = rule.declaration_specifiers__type_specifier($1);}
309     | type_qualifier declaration_specifiers {$$ = rule.declaration_specifiers__type_qualifier_declaration_specifiers();}
310     | type_qualifier {$$ = rule.declaration_specifiers__type_qualifier();}
311     | function_specifier declaration_specifiers {$$ = rule.declaration_specifiers__function_specifier_declaration_specifiers();}
312     | function_specifier {$$ = rule.declaration_specifiers__function_specifier();}
313     | alignment_specifier declaration_specifiers {$$ = rule.declaration_specifiers__alignment_specifier_declaration_specifiers();}
314     | alignment_specifier {$$ = rule.declaration_specifiers__alignment_specifier();}
315     ;
316
317 init_declarator_list
318     : init_declarator {$$ = rule.init_declarator_list__init_declarator($1);}
319     | init_declarator_list ',' init_declarator {$$ = rule.init_declarator_list__init_declarator_list_COMMA_init_declarator($1, $3);}
320     ;
321
322 init_declarator
323     : declarator '=' initializer {$$ = rule.init_declarator__declarator_EQ_initializer($1, $3);}
324     | declarator {$$ = rule.init_declarator__declarator($1);}
325     ;
326
327 storage_class_specifier
328     : TYPEDEF {$$ = rule.storage_class_specifier__TYPEDEF();}
329     | EXTERN {$$ = rule.storage_class_specifier__EXTERN();}
330     | STATIC {$$ = rule.storage_class_specifier__STATIC();}
331     | THREAD_LOCAL {$$ = rule.storage_class_specifier__THREAD_LOCAL();}
332     | AUTO {$$ = rule.storage_class_specifier__AUTO();}
333     | REGISTER {$$ = rule.storage_class_specifier__REGISTER();}
334     ;
335
336 type_specifier
337     : VOID {$$ = rule.type_specifier__VOID();}
338     | CHAR {$$ = rule.type_specifier__CHAR();}
339     | SHORT {$$ = rule.type_specifier__SHORT();}
340     | INT {$$ = rule.type_specifier__INT();}
341     | LONG {$$ = rule.type_specifier__LONG();}
342     | FLOAT {$$ = rule.type_specifier__FLOAT();}
```

```
343     | DOUBLE {$$ = rule.type_specifier__DOUBLE();}
344     | SIGNED {$$ = rule.type_specifier__SIGNED();}
345     | UNSIGNED {$$ = rule.type_specifier__UNSIGNED();}
346     | BOOL {$$ = rule.type_specifier__BOOL();}
347     | COMPLEX {$$ = rule.type_specifier__COMPLEX();}
348     | IMAGINARY {$$ = rule.type_specifier__IMAGINARY();}
349     | atomic_type_specifier {$$ = rule.type_specifier__atomic_type_specifier($1);}
350     | struct_or_union_specifier {$$ = rule.type_specifier__struct_or_union_specifier(↩
            $1);}
351     | enum_specifier {$$ = rule.type_specifier__enum_specifier($1);}
352     | TYPEDEF_NAME {$$ = rule.type_specifier__TYPEDEF_NAME($1);}
353     ;
354
355 struct_or_union_specifier
356     : struct_or_union '{' struct_declaration_list '}' {$$ = rule.↩
            struct_or_union_specifier__struct_or_union_LCB_struct_declaration_list_RCB()↩
            ;}
357     | struct_or_union IDENTIFIER '{' struct_declaration_list '}' {$$ = rule.↩
            struct_or_union_specifier__struct_or_union_IDENTIFIER_LCB_struct_declaration_list_RCB↩
            ();}
358     | struct_or_union IDENTIFIER {$$ = rule.↩
            struct_or_union_specifier__struct_or_union_IDENTIFIER();}
359     ;
360
361 struct_or_union
362     : STRUCT {$$ = rule.struct_or_union__STRUCT();}
363     | UNION {$$ = rule.struct_or_union__UNION();}
364     ;
365
366 struct_declaration_list
367     : struct_declaration {$$ = rule.struct_declaration_list__struct_declaration();}
368     | struct_declaration_list struct_declaration {$$ = rule.↩
            struct_declaration_list__struct_declaration_list_struct_declaration();}
369     ;
370
371 struct_declaration
372     : specifier_qualifier_list ';' {$$ = rule.↩
            struct_declaration__specifier_qualifier_list_SCOLON();}
373     | specifier_qualifier_list struct_declarator_list ';' {$$ = rule.↩
            struct_declaration__specifier_qualifier_list_struct_declarator_list_SCOLON()↩
            ;}
374     | static_assert_declaration {$$ = rule.↩
            struct_declaration__static_assert_declaration();}
375     ;
376
377 specifier_qualifier_list
378     : type_specifier specifier_qualifier_list {$$ = rule.↩
            specifier_qualifier_list__type_specifier_specifier_qualifier_list();}
379     | type_specifier {$$ = rule.specifier_qualifier_list__type_specifier();}
380     | type_qualifier specifier_qualifier_list {$$ = rule.↩
            specifier_qualifier_list__type_qualifier_specifier_qualifier_list();}
381     | type_qualifier {$$ = rule.specifier_qualifier_list__type_qualifier();}
382     ;
383
384 struct_declarator_list
385     : struct_declarator {$$ = rule.struct_declarator_list__struct_declarator();}
386     | struct_declarator_list ',' struct_declarator {$$ = rule.↩
            struct_declarator_list__struct_declarator_list_COMMA_struct_declarator();}
387     ;
388
```

```
389 struct_declarator
390     : ':' constant_expression {$$ = rule.struct_declarator__COLON_constant_expression↩
            ();}
391     | declarator ':' constant_expression {$$ = rule.↩
            struct_declarator__declarator_COLON_constant_expression();}
392     | declarator {$$ = rule.struct_declarator__declarator();}
393     ;
394
395 enum_specifier
396     : ENUM '{' enumerator_list '}' {$$ = rule.↩
            enum_specifier__ENUM_LCB_enumerator_list_RCB();}
397     | ENUM '{' enumerator_list ',' '}' {$$ = rule.↩
            enum_specifier__ENUM_LCB_enumerator_list_COMMA_RCB();}
398     | ENUM IDENTIFIER '{' enumerator_list '}' {$$ = rule.↩
            enum_specifier__ENUM_IDENTIFIER_LCB_enumerator_list_RCB();}
399     | ENUM IDENTIFIER '{' enumerator_list ',' '}' {$$ = rule.↩
            enum_specifier__ENUM_IDENTIFIER_LCB_enumerator_list_COMMA_RCB();}
400     | ENUM IDENTIFIER {$$ = rule.enum_specifier__ENUM_IDENTIFIER();}
401     ;
402
403 enumerator_list
404     : enumerator {$$ = rule.enumerator_list__enumerator();}
405     | enumerator_list ',' enumerator {$$ = rule.↩
            enumerator_list__enumerator_list_COMMA_enumerator();}
406     ;
407
408 enumerator
409     : enumeration_constant '=' constant_expression {$$ = rule.↩
            enumerator__enumeration_constant_EQ_constant_expression();}
410     | enumeration_constant {$$ = rule.enumerator__enumeration_constant();}
411     ;
412
413 atomic_type_specifier
414     : ATOMIC '(' type_name ')' {$$ = rule.↩
            atomic_type_specifier__ATOMIC_LP_type_name_RP();}
415     ;
416
417 type_qualifier
418     : CONST {$$ = rule.type_qualifier__CONST();}
419     | RESTRICT {$$ = rule.type_qualifier__RESTRICT();}
420     | VOLATILE {$$ = rule.type_qualifier__VOLATILE();}
421     | ATOMIC {$$ = rule.type_qualifier__ATOMIC();}
422     ;
423
424 function_specifier
425     : INLINE {$$ = rule.function_specifier__INLINE();}
426     | NORETURN {$$ = rule.function_specifier__NORETURN();}
427     ;
428
429 alignment_specifier
430     : ALIGNAS '(' type_name ')' {$$ = rule.↩
            alignment_specifier__ALIGNAS_LP_type_name_RP();}
431     | ALIGNAS '(' constant_expression ')' {$$ = rule.↩
            alignment_specifier__ALIGNAS_LP_constant_expression_RP();}
432     ;
433
434 declarator
435     : pointer direct_declarator {$$ = rule.declarator__pointer_direct_declarator($1, ↩
            $2);}
436     | direct_declarator {$$ = rule.declarator__direct_declarator($1);}
```

```
437     ;
438
439 direct_declarator
440     : IDENTIFIER {$$ = rule.direct_declarator__IDENTIFIER($1);}
441     | '(' declarator ')' {$$ = rule.direct_declarator__LP_declarator_RP($2);}
442     | direct_declarator '[' ']' {$$ = rule.↩
            direct_declarator__direct_declarator_LSB_RSB($1);}
443     | direct_declarator '[' '*' ']' {$$ = rule.↩
            direct_declarator__direct_declarator_LSB_MULT_RSB($1);}
444     | direct_declarator '[' STATIC type_qualifier_list assignment_expression ']' {$$ ↩
            = rule.↩
            direct_declarator__direct_declarator_LSB_STATIC_type_qualifier_list_assignment_expression_RSB↩
            ($1, $4, $5);}
445     | direct_declarator '[' STATIC assignment_expression ']' {$$ = rule.↩
            direct_declarator__direct_declarator_LSB_STATIC_assignment_expression_RSB($1,↩
             $4);}
446     | direct_declarator '[' type_qualifier_list '*' ']' {$$ = rule.↩
            direct_declarator__direct_declarator_LSB_type_qualifier_list_MULT_RSB($1, $3)↩
            ;}
447     | direct_declarator '[' type_qualifier_list STATIC assignment_expression ']' {$$ ↩
            = rule.↩
            direct_declarator__direct_declarator_LSB_type_qualifier_list_STATIC_assignment_expression_RSB↩
            ($1, $3, $5);}
448     | direct_declarator '[' type_qualifier_list assignment_expression ']' {$$ = rule.↩
            direct_declarator__direct_declarator_LSB_type_qualifier_list_assignment_expression_RSB↩
            ($1, $3, $4);}
449     | direct_declarator '[' type_qualifier_list ']' {$$ = rule.↩
            direct_declarator__direct_declarator_LSB_type_qualifier_list_RSB($1, $3);}
450     | direct_declarator '[' assignment_expression ']' {$$ = rule.↩
            direct_declarator__direct_declarator_LSB_assignment_expression_RSB($1, $3);}
451     | direct_declarator '(' parameter_type_list ')' {$$ = rule.↩
            direct_declarator__direct_declarator_LP_parameter_type_list_RP($1, $3);}
452     | direct_declarator '(' ')' {$$ = rule.direct_declarator__direct_declarator_LP_RP↩
            ($1);}
453     | direct_declarator '(' identifier_list ')' {$$ = rule.↩
            direct_declarator__direct_declarator_LP_identifier_list_RP($1, $3);}
454     ;
455
456 pointer
457     : '*' type_qualifier_list pointer {$$ = rule.↩
            pointer__MULT_type_qualifier_list_pointer($2, $3);}
458     | '*' type_qualifier_list {$$ = rule.pointer__MULT_type_qualifier_list($2);}
459     | '*' pointer {$$ = rule.pointer__MULT_pointer($2);}
460     | '*' {$$ = rule.pointer__MULT();}
461     ;
462
463 type_qualifier_list
464     : type_qualifier {$$ = rule.type_qualifier_list__type_qualifier($1);}
465     | type_qualifier_list type_qualifier {$$ = rule.↩
            type_qualifier_list__type_qualifier_list_type_qualifier($1, $2);}
466     ;
467
468
469 parameter_type_list
470     : parameter_list ',' ELLIPSIS {$$ = rule.↩
            parameter_type_list__parameter_list_COMMA_ELLIPSIS($1);}
471     | parameter_list {$$ = rule.parameter_type_list__parameter_list($1);}
472     ;
473
474 parameter_list
```

```
475     : parameter_declaration {$$ = rule.parameter_list__parameter_declaration($1);}
476     | parameter_list ',' parameter_declaration {$$ = rule.↩
            parameter_list__parameter_list_COMMA_parameter_declaration($1, $3);}
477     ;
478
479 parameter_declaration
480     : declaration_specifiers declarator {$$ = rule.↩
            parameter_declaration__declaration_specifiers_declarator();}
481     | declaration_specifiers abstract_declarator {$$ = rule.↩
            parameter_declaration__declaration_specifiers_abstract_declarator();}
482     | declaration_specifiers {$$ = rule.parameter_declaration__declaration_specifiers↩
            ();}
483     ;
484
485 identifier_list
486     : IDENTIFIER {$$ = rule.identifier_list__IDENTIFIER();}
487     | identifier_list ',' IDENTIFIER {$$ = rule.↩
            identifier_list__identifier_list_COMMA_IDENTIFIER();}
488     ;
489
490 type_name
491     : specifier_qualifier_list abstract_declarator {$$ = rule.↩
            type_name__specifier_qualifier_list_abstract_declarator();}
492     | specifier_qualifier_list {$$ = rule.type_name__specifier_qualifier_list();}
493     ;
494
495 abstract_declarator
496     : pointer direct_abstract_declarator {$$ = rule.↩
            abstract_declarator__pointer_direct_abstract_declarator();}
497     | pointer {$$ = rule.abstract_declarator__pointer();}
498     | direct_abstract_declarator {$$ = rule.↩
            abstract_declarator__direct_abstract_declarator();}
499     ;
500
501 direct_abstract_declarator
502     : '(' abstract_declarator ')' {$$ = rule.↩
            direct_abstract_declarator__LP_abstract_declarator_RP();}
503     | '[' ']' {$$ = rule.direct_abstract_declarator__LSB_RSB();}
504     | '[' '*' ']' {$$ = rule.direct_abstract_declarator__LSB_MULT_RSB();}
505     | '[' STATIC type_qualifier_list assignment_expression ']' {$$ = rule.↩
            direct_abstract_declarator__LSB_STATIC_type_qualifier_list_assignment_expression_RSB↩
            ();}
506     | '[' STATIC assignment_expression ']' {$$ = rule.↩
            direct_abstract_declarator__LSB_STATIC_assignment_expression_RSB();}
507     | '[' type_qualifier_list STATIC assignment_expression ']' {$$ = rule.↩
            direct_abstract_declarator__LSB_type_qualifier_list_STATIC_assignment_expression_RSB↩
            ();}
508     | '[' type_qualifier_list assignment_expression ']' {$$ = rule.↩
            direct_abstract_declarator__LSB_type_qualifier_list_assignment_expression_RSB↩
            ();}
509     | '[' type_qualifier_list ']' {$$ = rule.↩
            direct_abstract_declarator__LSB_type_qualifier_list_RSB();}
510     | '[' assignment_expression ']' {$$ = rule.↩
            direct_abstract_declarator__LSB_assignment_expression_RSB();}
511     | direct_abstract_declarator '[' ']' {$$ = rule.↩
            direct_abstract_declarator__direct_abstract_declarator_LSB_RSB();}
512     | direct_abstract_declarator '[' '*' ']' {$$ = rule.↩
            direct_abstract_declarator__direct_abstract_declarator_LSB_MULT_RSB();}
513     | direct_abstract_declarator '[' STATIC type_qualifier_list assignment_expression↩
            ']' {$$ = rule.↩
```

```
            direct_abstract_declarator__direct_abstract_declarator_LSB_STATIC_type_qualifier_list_assignment_
            ();}
514     | direct_abstract_declarator '[' STATIC assignment_expression ']' {$$ = rule.↩
            direct_abstract_declarator__direct_abstract_declarator_LSB_STATIC_assignment_expression_RSB↩
            ();}
515     | direct_abstract_declarator '[' type_qualifier_list assignment_expression ']' {↩
            $$ = rule.↩
            direct_abstract_declarator__direct_abstract_declarator_LSB_type_qualifier_list_assignment_express
            ();}
516     | direct_abstract_declarator '[' type_qualifier_list STATIC assignment_expression↩
             ']' {$$ = rule.↩
            direct_abstract_declarator__direct_abstract_declarator_LSB_type_qualifier_list_STATIC_assignment_
            ();}
517     | direct_abstract_declarator '[' type_qualifier_list ']' {$$ = rule.↩
            direct_abstract_declarator__direct_abstract_declarator_LSB_type_qualifier_list_RSB↩
            ();}
518     | direct_abstract_declarator '[' assignment_expression ']' {$$ = rule.↩
            direct_abstract_declarator__direct_abstract_declarator_LSB_assignment_expression_RSB↩
            ();}
519     | '(' ')' {$$ = rule.direct_abstract_declarator__LP_RP();}
520     | '(' parameter_type_list ')' {$$ = rule.↩
            direct_abstract_declarator__LP_parameter_type_list_RP();}
521     | direct_abstract_declarator '(' ')' {$$ = rule.↩
            direct_abstract_declarator__direct_abstract_declarator_LP_RP();}
522     | direct_abstract_declarator '(' parameter_type_list ')' {$$ = rule.↩
            direct_abstract_declarator__direct_abstract_declarator_LP_parameter_type_list_RP↩
            ();}
523     ;
524
525 initializer
526     : '{' initializer_list '}' {$$ = rule.initializer__LCB_initializer_list_RCB();}
527     | '{' initializer_list ',' '}' {$$ = rule.↩
            initializer__LCB_initializer_list_COMMA_RCB();}
528     | assignment_expression {$$ = rule.initializer__assignment_expression();}
529     ;
530
531 initializer_list
532     : designation initializer {$$ = rule.initializer_list__designation_initializer()↩
            ;}
533     | initializer {$$ = rule.initializer_list__initializer();}
534     | initializer_list ',' designation initializer {$$ = rule.↩
            initializer_list__initializer_list_COMMA_designation_initializer();}
535     | initializer_list ',' initializer {$$ = rule.↩
            initializer_list__initializer_list_COMMA_initializer();}
536     ;
537
538 designation
539     : designator_list '=' {$$ = rule.designation__designator_list_EQ();}
540     ;
541
542 designator_list
543     : designator {$$ = rule.designator_list__designator();}
544     | designator_list designator {$$ = rule.↩
            designator_list__designator_list_designator();}
545     ;
546
547 designator
548     : '[' constant_expression ']' {$$ = rule.designator__LSB_constant_expression_RSB↩
            ();}
549     | '.' IDENTIFIER {$$ = rule.designator__DOT_IDENTIFIER();}
```

```
550     ;
551
552 static_assert_declaration
553     : STATIC_ASSERT '(' constant_expression ',' STRING_LITERAL ')' ';' {$$ = rule.↩
            static_assert_declaration___STATIC_ASSERT_LP_constant_expression_COMMA_STRING_LITERAL_RP_SCO
            ();}
554     ;
555
556 statement
557     : labeled_statement {$$ = rule.statement___labeled_statement();}
558     | compound_statement {$$ = rule.statement___compound_statement();}
559     | expression_statement {$$ = rule.statement___expression_statement();}
560     | selection_statement {$$ = rule.statement___selection_statement();}
561     | iteration_statement {$$ = rule.statement___iteration_statement();}
562     | jump_statement {$$ = rule.statement___jump_statement();}
563     ;
564
565 labeled_statement
566     : IDENTIFIER ':' statement {$$ = rule.↩
            labeled_statement___IDENTIFIER_COLON_statement();}
567     | CASE constant_expression ':' statement {$$ = rule.↩
            labeled_statement___CASE_constant_expression_COLON_statement();}
568     | DEFAULT ':' statement {$$ = rule.labeled_statement___DEFAULT_COLON_statement();}
569     ;
570
571 compound_statement
572     : '{' '}' {$$ = rule.compound_statement___LCB_RCB();}
573     | '{' block_item_list '}' {$$ = rule.↩
            compound_statement___LCB___block_item_list_RCB();}
574     ;
575
576 block_item_list
577     : block_item {$$ = rule.block_item_list___block_item();}
578     | block_item_list block_item {$$ = rule.↩
            block_item_list___block_item_list_block_item();}
579     ;
580
581 block_item
582     : declaration {$$ = rule.block_item___declaration();}
583     | statement {$$ = rule.block_item___statement();}
584     ;
585
586 expression_statement
587     : ';' {$$ = rule.expression_statement___SCOLON();}
588     | expression ';' {$$ = rule.expression_statement___expression_SCOLON();}
589     ;
590
591 selection_statement
592     : IF '(' expression ')' statement ELSE statement {$$ = rule.↩
            selection_statement___IF_LP_expression_RP_statement_ELSE_statement();}
593     | IF '(' expression ')' statement {$$ = rule.↩
            selection_statement___IF_LP_expression_RP_statement();}
594     | SWITCH '(' expression ')' statement {$$ = rule.↩
            selection_statement___SWITCH_LP_expression_RP_statement();}
595     ;
596
597 iteration_statement
598     : WHILE '(' expression ')' statement {$$ = rule.↩
            iteration_statement___WHILE_LP_expression_RP_statement();}
599     | DO statement WHILE '(' expression ')' ';' {$$ = rule.↩
```

```
        iteration_statement__DO_statement_WHILE_LP_expression_RP_SCOLON();}
600     | FOR '(' expression_statement expression_statement ')' statement {$$ = rule.↩
        iteration_statement__FOR_LP_expression_statement_expression_statement_RP_statement↩
        ();}
601     | FOR '(' expression_statement expression_statement expression ')' statement {$$ ↩
        = rule.↩
        iteration_statement__FOR_LP_expression_statement_expression_statement_expression_RP_statement↩
        ();}
602     | FOR '(' declaration expression_statement ')' statement {$$ = rule.↩
        iteration_statement__FOR_LP_declaration_expression_statement_RP_statement();}
603     | FOR '(' declaration expression_statement expression ')' statement {$$ = rule.↩
        iteration_statement__FOR_LP_declaration_expression_statement_expression_RP_statement↩
        ();}
604     ;
605
606 jump_statement
607     : GOTO IDENTIFIER ';' {$$ = rule.jump_statement__GOTO_IDENTIFIER_SCOLON();}
608     | CONTINUE ';' {$$ = rule.jump_statement__CONTINUE_SCOLON();}
609     | BREAK ';' {$$ = rule.jump_statement__BREAK_SCOLON();}
610     | RETURN ';' {$$ = rule.jump_statement__RETURN_SCOLON();}
611     | RETURN expression ';' {$$ = rule.jump_statement__RETURN_expression_SCOLON();}
612     ;
613
614 translation_unit
615     : external_declaration {$$ = rule.translation_unit__external_declaration($1);}
616     | translation_unit external_declaration {$$ = rule.↩
        translation_unit__translation_unit_external_declaration($1, $2);}
617     ;
618
619 external_declaration
620     : function_definition {$$ = rule.external_declaration__function_definition($1);}
621     | declaration {$$ = rule.external_declaration__declaration($1);}
622     ;
623
624 function_definition
625     : declaration_specifiers declarator declaration_list compound_statement {$$ = ↩
        rule.↩
        function_definition__declaration_specifiers_declarator_declaration_list_compound_statement↩
        ();}
626     | declaration_specifiers declarator compound_statement {$$ = rule.↩
        function_definition__declaration_specifiers_declarator_compound_statement();}
627     ;
628
629 declaration_list
630     : declaration {$$ = rule.declaration_list__declaration();}
631     | declaration_list declaration {$$ = rule.↩
        declaration_list__declaration_list_declaration();}
632     ;
633
634 %%
635 #include <stdio.h>
636
637 void yyerror(const char *s)
638 {
639     fflush(stdout);
640     fprintf(stderr, "*** %s\n", s);
641 }
```