Universidade FUMEC

Faculdade de Ciências Empresariais

Programa de Pós-Graduação em Sistemas de Informação e Gestão do Conhecimento

Lauro Henrique Timóteo de Oliveira

Desafios da automatização de processos de migração de softwares

Área de Concentração

SISTEMAS DE INFORMAÇÃO E GESTÃO DO CONHECIMENTO

Linha de Pesquisa

TECNOLOGIA E SISTEMAS DE INFORMAÇÃO

Belo Horizonte

2018

Lauro Henrique Timóteo de Oliveira

Desafios da automatização de processos de migração de softwares

Dissertação apresentada ao Programa de Mestrado em Sistemas de Informação e Gestão do Conhecimento da Universidade Fundação Mineira de Educação e Cultura — FUMEC, como requisito parcial para a obtenção do título de Mestre em Sistemas de Informação e Gestão do Conhecimento.

Área de concentração: Gestão de Sistemas de Informação e Conhecimento.

Linha de pesquisa: Tecnologia e Sistemas de Informação

Professor Orientador: Luiz Cláudio Gomes Maia

Belo Horizonte

2018

Dados Internacionais de Catalogação na Publicação (CIP)

O48d Oliveira, Lauro Henrique Timóteo de, 1983 -
Desafios da automatização de processos de migração de softwares / Lauro Henrique Timóteo de Oliveira. – Belo Horizonte, 2018.
124 f : il. ; 29,7 cm

Orientador: Luiz Cláudio Maia
Dissertação (Mestrado em Sistemas de Informação e Gestão do Conhecimento), Universidade FUMEC, Faculdade de Ciências Empresariais, Belo Horizonte, 2018.

1. Software - Brasil. 2. PowerBuilder (Programa de computador). 3. C (Linguagem de programação de computador). I. Título. II. Maia, Luiz Cláudio. III. Universidade FUMEC, Faculdade de Ciências Empresariais.

CDU: 65.01:001

Ficha catalográfica elaborada pela Biblioteca Universitária-FUMEC

UNIVERSIDADE
FUMEC

Dissertação intitulada "**Desafios da automatização de processos de migração de softwares**" de autoria de Lauro Henrique Timóteo de Oliveira, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Luiz Cláudio Gomes Maia – Universidade FUMEC
(Orientador)

Prof. Dr. Daniel Jardim Pardini – Universidade FUMEC
(Examinador Interno)

Prof. Dr. Alair Dias Júnior – UFMG
(Examinador Externo)

Thiago Campos Pereira, Esp. – Senior Solunion
(Consultor *Ad Hoc*)

Prof. Dr. Fernando Silva Parreiras
Coordenador do Programa de Pós-Graduação em Sistemas de Informação e Gestão do Conhecimento da Universidade FUMEC

Belo Horizonte, 05 de fevereiro de 2018.

REITORIA CAMPUS

Dedico esse trabalho aos meus pais, Luiz Carlos e Vânia, à minha esposa Ana Paula, e aos meus filhos, Gabriela e Rafael, que tudo faço por eles.

AGRADECIMENTOS

Ao professor e orientador Luiz Claudio Maia, pelas primeiras aulas ainda como disciplina isolada que me fizeram encantar novamente pela busca de novos conhecimentos.

A antiga attps informática, atualmente Senior Solution, pelo apoio financeiro e pela liberdade durante a execução da pesquisa.

À professora Leia Garcia pela oportunidade e confiança que me levaram a iniciar a carreira acadêmica e consequentemente à mais esta conquista.

Aos professores do programa pelos ensinamentos, paciência e companheirismo.

Aos meus pais que nunca pouparam esforços com minha educação. Sempre presentes no dia a dia da minha educação e formação do meu caráter.

À minha irmã Carolina, pela força e ajuda em momentos difíceis durante o curso. Irmãzinha esse título também é seu!

Aos meus amigos que em momentos difíceis me ajudaram, deram apoio e mantiveram firme nessa caminhada.

A minha esposa, que me suportou durante todo este tempo, dando apoio e me guiando nas horas em que me perdia.

Aos meus filhos, que sentiram à minha ausência, mas faço por vocês, amo vocês!

Resumo

Há anos sistemas são construídos sobre plataformas de desenvolvimento. Assim como a tecnologia da informação, as plataformas de desenvolvimento evoluem muito rápido, trazendo ganhos para os negócios que muitas vezes não podem ser aproveitados pelos sistemas legados, feitos em plataformas mais antigas. Este trabalho aplica um processo automatizado de migração de códigos fontes escritos sobre a plataforma PowerBuilder para a linguagem C#.net em uma empresa de desenvolvimento de software. Foi construído uma ferramenta que faz a conversão automatizada do código utilizando técnicas de compiladores que foi capaz de converter 99% das linhas de código sem erros. O código fonte gerado foi avaliado através de questionários e workshop e obteve avaliação positiva quanto a qualidade código. Este trabalho ainda apresenta cálculos para avaliação da viabilidade financeira de processos desta natureza, que neste caso se demostrou economicamente viável gerando mais de 2.000.000 (dois milhões de reais) em economia.

Palavras-chave: modernização de software, migração de código, automatização, C#, PowerBuilder.

Abstract

For years systems are built on many development platforms. As well as information technology, development platforms evolve very quickly, bringing gains for the business that often cannot be used by legacy systems, made on older platforms. This work applies an automated process of migrating source code written on PowerBuilder platform to C# .net language in a software development company. An automated code conversion tool using compiler techniques was built and it was able to convert 99% of line codes without errors. The generated source code was evaluated through survey and workshop that proved satisfactory quality. This work still presents calculations for evaluating the financial viability of processes of this nature, which in this case proved economically viable, saving more than two million reais (R$ 2.000.000,00) of project costs.

Keywords: software modernization, code migration, automation, C#, PowerBuilder

Sumário

1 INTRODUÇÃO 15

1.1 Motivação e justificativa 18

1.2 Objetivos 19

2 FUNDAMENTAÇÃO TEÓRICA 20

2.1 Modernização de software 20

2.1.1 Estratégias de modernização de software 21

2.1.2 Rejuvenescimento de código 22

2.2 Teoria de compiladores 23

2.2.1 Análise do programa fonte 25

2.2.2 Geração do programa alvo 36

2.3 Tamanho de sistemas 37

2.3.1 Tamanho funcional 38

2.4 Tecnologias e ferramentas 39

2.4.1 A linguagem C# 39

2.4.2 A Plataforma PowerBuilder 42

2.4.3 O Compilador Roslyn 42

2.4.4 ANTLR 43

2.5 Trabalhos relacionados 44

3 METODOLOGIA 47

3.1 Considerações iniciais 47

3.2 Etapas da Pesquisa 47

3.3 Construção do compilador 48

3.3.1 Análise 48

3.3.2 Geração 49

3.3.3 As etapas do compilador 51

3.4 Conversão Automatizada 58

3.4.1 remcli 60

3.4.2 sfrintsac 61

3.5 Workshop 62

3.5.1 Qualidade do código fonte gerado 63

3.5.2 Estimativa de intervenção manual 64

3.6 Síntese dos desafios 65

3.6.1 Classificações dos Desafios 65

3.6.2 Desafios de Análise Léxico-sintática 67

3.6.3 Desafios de Transformação 71

3.7 Avaliação da viabilidade da automação 84

4 Resultados 92

4.1.1 Construção do compilador 92

4.1.2 Conversão Automatizada 92

4.1.3 Workshop 93

4.1.4 Síntese dos desafios 95

4.1.5 Avaliação da viabilidade da automação 97

5 Conclusão 98

5.1 Limitações da Pesquisa 99

5.2 Trabalhos futuros 100

ANEXO I 102

ANEXO II 112

Referências 119

Lista de Figuras

Figura 1- Um compilador 24

Figura 2 - Árvore sintática 24

Figura 3 - Árvore sintática da gramática G expressão 2 + 1 31

Figura 4 - Estimativa e medição – 38

Figura 5 – Trifurcação do .Net Framework 41

Figura 6 - .Net Standard 41

Figura 7- Estrutura das APIs do Roslyn 43

Figura 8- Etapas x Objetivos 47

Figura 9 - Etapas do projeto 47

Figura 10 - Fluxo de conversão do código fonte PowerBuilder para C# 50

Figura 11 - Fluxo de Execução do Compilador 51

Figura 12 - Estrutura de código fonte PowerBuilder 52

Figura 13 - Coerção automática do PowerBuilder 55

Figura 14 - Marcação semântica de uma AST 56

Figura 15 - Exemplo de código convertido 57

Figura 16 - Relação de erros de compilação no fonte migrado da remcli 61

Figura 17- Resultado da compilação do fonte migrado da rotina sfrintsac 62

Figura 18 - Fragmentos lexicos para definir keywords como case insensitive 68

Figura 19 - Construção léxica de algumas Keywords do SQL 68

Figura 20- Regra léxica com intervenção de código para resolução de ambiguidades 69

Figura 21 - Definindo Line Continuing para canal oculto (Hidden) 70

Figura 22 - Definição lexica para comentários de linha resolvendo ambuiguidade do delimitador de comandos 71

Figura 23 - Exemplo de transformação do Choose Case 72

Figura 24 - Exemplo de transformação do SQL Embbeded 73

Figura 25- Exemplo de transformação do 1 Based Array 74

Figura 26 - Exemplo de transformação do Initial Value of Variables 75

Figura 27 - Exemplo de transformação do File Manipulation 76

Figura 28 – Inclusão de classe SystemFunction como importação estática 77

Figura 29 - Exemplo de transformação do Global User Functions 77

Figura 30 - Exemplo de transformação do Transaction Class 79

Figura 31 - Exemplo de transformação do Automatic Number Casting 80

Figura 32 - Exemplo de transformação do Char and String Literals 81

Figura 33 - Exemplo de transformação do Global Variables 83

Figura 34 - Exemplo de transformação do Exponential Operator 83

Figura 35- Tela de consulta de apontamento de horas 86

Figura 36- Relatório de apontamento de horas 87

Figura 37- Desmembramento do cálculo de viabilidade por linhas de código 90

Lista de tabelas

Tabela 1 - Resultado análise dos fontes para eleição de migração 59

Tabela 2 - Nível de Senioridade dos programadores participantes do Workshop 62

Tabela 3 - Esforço para intervenção manual de itens não migrados 95

Tabela 4- Relação entre tipos inteiros do PowerBuilder e C# 84

Lista de quadros

Quadro 1 - Resultado consolidado da análise dos fontes de programas candidatos a migração 59

Quadro 2- Indicadores análise processo 85

Quadro 3 - Refinamento dos indicadores 87

Quadro 5 - Valor dos indicadores coletados dos sistemas de gestão............................88

Quadro 6 - Funcionalidades não migradas 93

Quadro 7-Resultado analítico da pesquisa de qualidade do fonte migrado..................94

Quadro 8 - Resultado consolidado da pesquisa de qualidade do fonte migrado 94

Quadro 9 – Relação de desafios com outras linguagens origem 97

Lista de siglas

Sigla	**Descrição**
CLI	Common Language Infrastucture
IDE	Integrated Development Envirionment
POO	Programação Orientada à Objetos
FPA	Análise de Ponto de Função
AST	Arvore de Sintaxe Abstrata
PBL	PowerBuilder Library
LOC	linhas de código
DW	DataWindo
WMU	Warrants, Maintenance, Upgrade
SABA	Software as Busness Asset
OMG	Object Management Group
ADM	Architecture Driven Modernization
VDM	Visaggio’s Decision Model
BNF	Forma de Backus-Naur
JIT	Just In Time
MSIL	Microsoft Intermediate Language
API	Application Programming Interface
PDD	provisão de devedores duvidosos
DML	Data Maintaining Language
DRL	Data Retrieve Language

1 INTRODUÇÃO

Manutenção e evolução de softwares estão entre 50 e 75 por cento do custo do software durante sua vida útil, em casos de softwares de longa duração (KOSKINEN, 2005). De acordo com a primeira lei de Lehman (1998) o software deve ser continuamente adaptado ou se tornará progressivamente menos satisfatório em ambientes “do mundo real”.

Sistemas legados são quaisquer sistemas críticos ao *core business* das organizações que resistem à modificações e suas falhas podem causar sérios impactos nos negócios (Brodie & Stonebraker, 1995). Esses sistemas podem causar diversos problemas para suas organizações (Bisbal, Lawless, Bing Wu, & Grimson, 1999):

- Sistemas legados rodam normalmente sobre plataformas obsoletas cujo hardware é lento e difícil de manter.
- Manutenção é mais cara, por falta de documentação e profissionais no mercado
- Dificuldade de integrar com outras plataformas mais recentes
- Difícil, ou impossível, de se estender.

Muitos sistemas legados tiveram anos de investimentos e contém imensuráveis regras de negócio e conhecimento em suas linhas de código. Estes sistemas são executados em ambientes obsoletos que precisam integrar-se com sistemas mais novos. Estas linhas de código de sistemas legados carecem muitas vezes de um processo de modernização para possibilitar a integração com novos ambiente e plataformas (KOSKINEN, 2005).

Linguagens de programação evoluem com o passar do tempo. Um dos direcionadores desta evolução é a simplificação do uso destas linguagens em projetos da vida real (PIRKELBAUER, 2010). Algumas linguagens se tornam obsoletas e deixam de evoluir. Neste caso um processo de modernização requererá a troca da linguagem ao qual o sistema é codificado.

A TIOBE, empresa de qualidade de software, libera mensalmente um ranking de popularidade de linguagens de programação. Quanto mais popular uma linguagem mais evoluções esta terá e mais profissionais disponíveis existirão no mercado, o que impacta diretamente os custos de mão de obra. Para o cálculo são usadas informações sobre quantidade de profissionais certificados, cursos oferecidos e estatísticas de buscas nos principais buscadores da internet: Google, Bing, Yahoo, Wikipedia, Amazon, Youtube e Baidu. Linguagens obsoletas como COBOL e Fortran se encontram nas posições 27 e 28 respectivamente e linguagens populares nos anos 90 como FoxPro, WinSQL e PowerBuilder (PowerScript) não estão entre as 50 primeiras (TIOBE, 2016).

Normalmente modernização de software gera modificações em diversas partes dos sistemas demandando alto custo de mão de obra. Um processo manual de migração de código fonte é caro, demorado e suscetível a erros (PIRKELBAUER, 2010).

Um software pode se tornar obsoleto por questões funcionais ou tecnológicas. Existem duas visões diferentes a respeito de um software obsoleto. Existe a visão do usuário e a visão do produtor ou fornecedor do software. O usuário pode optar por: continuar a utilizar o software ou procurar um outro software de outro fornecedor caso exista. O produtor ou fornecedor de software pode optar por: remover o software parcialmente ou totalmente do mercado, reescrever totalmente o software ou modernizá-lo (KOSKINEN, 2005). Neste trabalho é abordada a visão do produtor do software no que tange à obsolescência de seus produtos.

Processos de modernização podem ocorrer de diversas formas, Fontanette (2004) em sua dissertação de mestrado descreve um processo manual gradativo, já (PIRKELBAUER, 2010) em sua tese de doutorado, onde descreve um método de rejuvenescimento de código fonte, sugere um processo automatizado.

Outros autores como Chisolm & Lisonbee (1999), Fujiwara, Ishiura, Sakai, Aoki, & Ogawara (2016) e NEWCOMB & DOBLAR (2001) demonstram processo automatizados para modernização do software.

CHISOLM & LISONBEE (1999) tomam como premissa que o *hardware* dos computadores serão substituídos e causarão impactos significantes nos sistemas legados relacionados a estes *hardwares.* No trabalho apresenta as vantagens de se utilizar técnicas de projeto de compiladores como solução para migração de código de sistemas legados.

FUJIWARA et al., (2016) apresentam um método de construir programas em linguagem C a partir de *assembly* de programas de *mainframe* IBM. Estes subprogramas são chamados normalmente de linguagens de níveis mais altos como COBOL e são convertidos em linguagem C que é mais portável para outras plataformas. O artigo ainda demostra a técnica utilizada para a tradução dos programas utilizando uma representação intermediária do programa de origem.

NEWCOMB & DOBLAR (2001) partem da premissa que atualmente já é possível fazer a migração automatizada de códigos fontes em níveis superiores a 99% (noventa e nove por cento). Eles descrevem um processo de modernização de software baseado em automação que alcançam níveis de conversões superiores a 99% de linhas de código, índices também alcançados neste trabalho.

A tecnologia serve de suporte para o negócio das organizações. Atualizações tecnológicas são constantemente preteridas em relação a atualizações funcionais, o que é totalmente justificável. Este processo torna em algum momento o software obsoleto sendo executado em plataformas não mais suportadas pelos seus fornecedores.

A modernização de software pode ocorrer de forma gradativa ou *ad-hoc* (todo o software de uma vez). Mais aplicada pelas empresas, a estratégia *ad-hoc* traz como desafio e risco manter a versão em processo de modernização coerente com a versão em produção pois o software legado na maior parte das vezes continua sofrendo manutenções exigidas pelo

negócio da organização. O processo gradativo permite a evolução em partes menores, porém pode não se aplicar quando a mudança de plataforma for muito drástica, ou quando as partes do sistema são muito acopladas.

Independente da forma aplicada de modernização (*ad-hoc* ou gradativa) pode-se utilizar de automação. Para a migração de grande quantidade de código é necessário um compilador (Chisolm & Lisonbee, 1999). Neste trabalho foi construído um compilador e as principais dificuldades foram catalogadas assim como o procedimento utilizado para superá-las.

1.1 Motivação e justificativa

Existem diversos compiladores e migradores de código comerciais e de código livre que possibilitam e até mesmo facilitam a criação de processos automatizados de migração de código fonte (Cifuentes, Simon, & Fraboulet, 1998; El-Ramly, Eltayeb, & Alla, 2006; Fujiwara et al., 2016).

Porém linguagens pouco populares como: PowerBuilder, SQLWindows, FoxPro não possuem tantos recursos (TIOBE, 2016), assim o processo de migração de código destas linguagens para linguagens mais populares depende da criação de um compilador que o faça.

A cada dia a construção de compiladores fica mais complexa devido a evolução das arquiteturas computacionais e a evolução das linguagens de programação (K.V.N. Sunitha, 2013). Este trabalho enumera as dificuldades encontradas, e como resolvê-las, para a construção de um compilador com foco na migração de código.

O processo de migração do código fonte manual é caro, demorado e suscetível a erros (PIRKELBAUER, 2010). Para a migração automatizada de grandes quantidades de código é necessário um compilador (Chisolm & Lisonbee, 1999). A construção de compiladores está cada vez mais complexa (K.V.N. Sunitha, 2013), o que justificou a execução deste trabalho e dimensiona o quanto este pode contribuir para o aumento do conhecimento nesta área.

Os problemas supracitados podem ser sumarizados da seguinte questão de pesquisa:

“Quais os desafios de um processo automatizado de migração de código fonte de sistemas legados?”

A seguir vamos apresentar os objetivos desta pesquisa de forma a responder à pergunta citada acima.

1.2 Objetivos

Este trabalho se propõe a identificar os desafios de um processo automatizado de migração de código fonte. Para alcançar este objetivo foram definidos os objetivos específicos a seguir:

1. Construir um compilador que será utilizado como ferramenta para o processo de migração de código.
2. Criar uma síntese dos desafios envolvidos no processo automatizado de migração de código fonte.
3. Validar a qualidade do código fonte gerado pelo compilador.
4. Obter uma estimativa do esforço de intervenção manual referente as funcionalidades não migradas pelo compilador.
5. Analisar a viabilidade econômica do processo automatizado.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão abordados conceitos que serão discutidos no decorrer desta dissertação assim como apresentação de técnicas e ferramentas utilizadas na execução da pesquisa aplicada. No capítulo 3 onde será descrita a metodologia os conceitos exibidos neste capítulo serão utilizados exaustivamente.

A seção 2.1 irá descrever os conceitos básicos da modernização de software assim como tipos e técnicas. A seção 2.2 apresenta conceitos sobre teoria de compiladores que serão utilizados para a criação da ferramenta de migração de código. Na seção 2.3 são apresentadas ferramentas e tecnologias que serão utilizadas nesta pesquisa. A seção 2.4 serão apresentados alguns trabalhos relacionados que servem de suporte para esta pesquisa.

2.1 Modernização de software

A modernização de software se refere ao processo de reescrever ou portar um sistema legado para uma linguagem de programação, plataforma, protocolo, plataforma de hardware ou biblioteca de software mais recente. (Koskinen, Ahonen, Lintinen, Sivula, & Tilus, 2004).

A modernização de software é necessária quando a desatualização tecnológica do software em questão passa a comprometer a operação ao qual o software se propõe, ou quando o custo de manutenção nesta plataforma se eleva a quantidades significativas ou se projetam de forma a se tornar risco ao retorno financeiro ao qual o software se propõe.

A modernização pode ocorrer através da reengenharia completa de um sistema ou apenas a atualização em partes. Algumas estratégias permitem reconstrução gradativa do software mantendo integração entre as partes modernizadas com partes ainda pendentes de modernização. Outras requerem um processo completo feito de uma única vez.

2.1.1 Estratégias de modernização de software

Existem diversas estratégias de modernização de software e cada uma se aplica a uma situação distinta onde trará benefícios. Na literatura existem diversas estratégias algumas listadas a seguir:

1. SAHIN & ZAHEDI, (2001) descrevem o modelo WMU (Warrants, Maintenance, Upgrade) para escolher estratégias de manutenção baseadas na expectativa de satisfação do cliente. Eles identificaram uma classe otimizada de políticas, próxima do ideal, e robustas que tornaram mais fácil a implementação e relação da efetividade dessas políticas em diferentes condições de mercado.
2. BENNETT, RAMAGE & MUNRO (1999) descrevem o modelo SABA que propõe um framework de alto nível que permite planejar a evolução dos sistemas legados levando em consideração questões organizacionais e tecnológicas. Eles demonstram a execução de um planejamento de modernização segundo os critérios definido do framework SABA. Os resultados são satisfatórios no caso apresentado.
3. WARREN & RANSOM (2002) descrevem o método Renaissance que permite avaliar interativamente sistemas legados, através das perspectivas técnicas, de negócios e organizacionais. O método propões uma análise por engenharia reversa para obter uma base estável do sistema e então continuamente expandir o sistema através de um fluxo de modificações incrementais. Os autores concluíram que o método aumenta a relação custo benefício do processo reduzindo também os seus riscos.
4. ULRICH (2004) através da OMG (Object Management Group) propõe o ADM (Architecture Driven Modernization), uma iniciativa de padronizar visualizações de sistemas existentes de forma a possibilitar estratégias comuns

de modernização de software, como análise e compreensão de código e transformação de software. O artigo ainda relata detalhes da força tarefa feita utilizando o ADM para guiar o desenvolvimento de uma série de padrões de modernização.

5. VISAGGIO (2000) propõe o VDM (Visaggio's Decision Model) onde é descrito um modelo de decisão para renovação de software no nível de componentes baseando-se nas qualidades técnicas e econômicas destes. O modelo de decisão proposto pode ser usado para projetos de modernização, mas o artigo também demostra que ele representa um pacote de experiências e que deve ser continuamente aperfeiçoado. O autor ainda demostra dois problemas em aberto: um é a validação externa dos resultados e o outro a falta de uma pesquisa que possibilite estimar o custo de processo de modernização.

2.1.2 Rejuvenescimento de código

Rejuvenescimento de código fonte é a transformação de fonte-para-fonte que substitui funcionalidades e comandos depreciados por código moderno. O código rejuvenescido é mais conciso, seguro e usa níveis mais altos de abstração. O rejuvenescimento de código é um processo unidirecional que detecta códigos expressos em instruções de baixo nível e os converte para níveis mais altos de abstração. (PIRKELBAUER, 2010)

Um processo de modernização de software permite a evolução técnica do código fonte. Este preza, além de atualizar o código, melhorá-lo utilizando construções mais otimizadas.

O conceito de rejuvenescimento de código fonte pode ser facilmente confundido com o conceito de refatoração de código que preza pela reorganização do código fonte. O processo de modernização de software pode conter ambas as técnicas, porém apenas refatoração não é capaz de modernizar um software.

PIRKELBAUER (2010) propõe que a modernização de software seja automatizada, feita por processos computacionais que interpretam o código e modificam-no de forma a torná-lo mais atualizado.

PIRKELBAUER (2010) ainda demostra em sua tese o processo atualizando construções idiomáticas de uma linguagem. Este mesmo modelo proposto por ele pode ser utilizado para traduzir ou transformar o código de linguagens obsoletas para novas linguagem e plataformas.

2.2 Teoria de compiladores

Compiladores são ferramentas de tradução entre linguagens, mantendo a semântica original. Lê-se uma linguagem e reescreve em outra. Tradicionalmente compiladores leem códigos escritos em uma linguagem de programação em alto nível e reescrevem aquele mesmo código em linguagem de máquina (K.V.N. Sunitha, 2013). Compiladores também podem ser utilizados para tradução entre linguagens de alto nível.

ULLMAN, AHO & SETHI (1995) definem um compilador de forma simples: “um compilador é um programa que lê um programa escrito numa linguagem – a linguagem fonte – e o traduz num programa equivalente numa outra linguagem – a linguagem alvo”.

Existe uma variedade de compiladores devido à variedade de linguagens fonte e linguagens alvo. Cada relação linguagem fonte x linguagem alvo necessita de um compilador especializado para esta relação. Apesar de diversificados, compiladores em geral fazem sempre as mesmas tarefas básicas, análise e síntese.

Compiladores também podem ser classificados como de uma passagem ou de passagens múltiplas. Dependendo de particularidades da linguagem fonte não é possível com apenas uma passagem (leitura do código fonte) executar as operações necessárias para a geração do código na linguagem alvo. A Figura 1 mostra um modelo básico de um compilador.

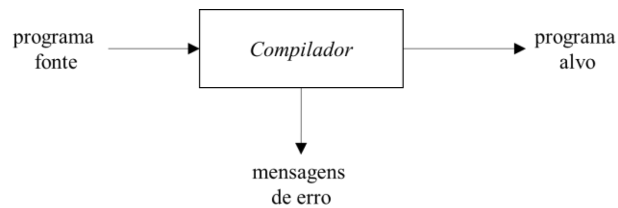

Figura 1- Um compilador

O compilador processa o programa fonte em sua primeira tarefa básica: a análise. Esta etapa separa o programa fonte em partes criando uma representação intermediária (ainda não é a linguagem alvo). Finalizada a análise inicia-se a segunda tarefa básica: a síntese. Esta etapa constrói o programa alvo desejado, a partir da representação intermediária gerada pela síntese. (Ullman et al., 1995).

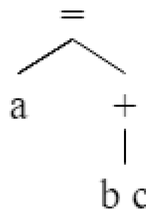

Figura 2 - Árvore sintática
Fonte: elaborado pelo autor

Durante a análise o código da linguagem fonte é separado em estruturas e organizado em forma de árvore. Nesta árvore denominada árvore sintática cada operação é incluída em um nó e seus argumentos (operação ou não) são incluídos como filhos deste nó.

A Figura 2 demonstra um exemplo de uma árvore sintática para uma expressão de atribuição de uma soma (a = b + c). Desta forma o código escrito na linguagem fonte se encontra estruturado de forma mais fácil de ser processada computacionalmente. Linguagens de programação são mais fáceis de serem compreendidas por humanos do que por máquinas assim a necessidade de transformar o código descrito na linguagem fonte em uma estrutura organizada (Appel & Ginsburg, 1997).

2.2.1 Análise do programa fonte

A análise do programa fonte é dividida em três fases: análise linear (análise léxica), análise hierárquica (análise sintática) e análise semântica. A análise linear faz a análise de cada caractere que constitui o programa fonte (da esquerda para direita) agrupando-os em *tokens* que são sequências de caracteres que possuem um significado na linguagem fonte (Parr, 2009). A análise hierárquica, agrupa os *tokens* hierarquicamente em forma de árvore (árvore sintática). A análise semântica garante a combinação correta dos elementos na árvore sintática e consegue dar significado forte a cada nó da árvore sintática.

Todas as três fases da análise são baseadas nas regras definidas para a linguagem fonte. Uma linguagem é definida por sua gramática. Uma gramática define basicamente qual são as construções existentes em uma linguagem (K.V.N. Sunitha, 2013).

2.2.1.1 Gramática

Uma gramática é como um conjunto finito de regras que quando aplicadas sucessivamente geram palavras. O Conjunto de todas as palavras geradas por uma gramática define uma linguagem (Menezes, 2011).

ULLMAN (1995) define gramática como regras que descrevem a estrutura sintática de uma linguagem de programação.

Uma gramática de Chomsky, gramática irrestrita ou simplesmente gramática é uma quádrupla ordenada (Chomsky, 1955):

$$G=(V, T, P, S)$$

Onde:

- **V**, um conjunto finito de símbolos variáveis ou não terminais;
- **T**, um conjunto finito de símbolos terminais disjuntos de V

- **P**: $\mathbf{P}: (\boldsymbol{V} \cup \boldsymbol{T})^{+} \longrightarrow (\boldsymbol{V} \cup \boldsymbol{T})^{*}$ é uma relação finita (ou seja, **P** é um conjunto finito de pares), denominada de relação de produções ou simplesmente produções.
- **S**, é um elemento distinguido de **V** denominado símbolo inicial ou variável inicial.

As regras de produção de uma gramática definem a forma como as construções são variadas a partir de símbolos não terminais. Cada símbolo não terminal é definido deforma a ser derivado até uma sequência de símbolos terminais. O exemplo a seguir define uma gramática G que descreve uma linguagem para escrever operações matemáticas de subtração, adição, multiplicação e divisão.

G = ({expr, num, dig}, {+,-,x,÷,1,2,3,4,5,6,7,8,9,0, (,)},P, expr)

P = {

expr → expr + expr

expr → expr - expr

expr → expr x expr

expr → expr ÷ expr

expr → (expr)

expr → num

num → dig

num → num dig

dig → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

}

A gramática G possui três símbolos variáveis ou não terminais: expr, num, dig. Possui 16 símbolos terminais, possui um conjunto de regras de produção **P**, que descreve as

construções possíveis na gramática e como as variáveis podem ser derivadas, e o símbolo “expr” é o símbolo inicial ou variável inicial.

Analisando as regras de produção da gramática G é possível identificar a função de cada símbolo. Os símbolos terminais denotam dígitos e operadores já os símbolos não terminais possuem um significado maior onde “expr” define uma expressão, “num” um número e “dig” define o que pode ser utilizado como dígito. Nota-se também que um número pode ser um único dígito ou conjunto de dígitos e que uma expressão pode ser um número, ou uma outra expressão entre parênteses além de poder ser uma operação binária de divisão, multiplicação, subtração ou adição. A expressão “(3 + 2) x 4” é uma sequência válida para a linguagem descrita pela gramática G (L(G)). O processo de derivação da gramática é feito iniciando-se do símbolo inicial, no caso da gramática G o símbolo “expr”.

expr	deriva para	expr x expr
expr x expr	deriva para	expr x num
expr x num	deriva para	expr x dig
expr x dig	deriva para	expr x 4
expr x 4	deriva para	(expr) x 4
(expr) x 4	deriva para	(expr + expr) x 4
(expr + expr) x 4	deriva para	(expr + num) x 4
(expr + num) x 4	deriva para	(expr + dig) x 4
(expr + dig) x 4	deriva para	(expr + 2) x 4
(expr + 2) x 4	deriva para	(num + 2) x 4
(num + 2) x 4	deriva para	(dig + 2) x 4
(dig + 2) x 4	deriva para	(3 + 2) x 4

A derivação finaliza quando não existem mais símbolos não terminais. Este processo é o mesmo utilizado pelo analisador sintático, símbolo não variável gera uma ramificação na árvore sintática e cada símbolo terminal gera uma folha.

A notação de CHOMSKY (1955) permite fácil compreensão do funcionamento das gramáticas, porém existem outras forma mais comumente utilizadas para descrição de gramática. Outras formas de definição de gramática permitem geração automática de analisadores léxicos na construção de compiladores como a BNF (Forma de Backus-Naur). A notação BNF permite descrever a gramática de forma simular a definição de CHOMSKY, (1955). A principal diferença é que os símbolos não precisam ser definidos previamente, e são definidos à medida que são utilizados na gramática. É possível também em algumas variações utilizar regras básicas de expressões regulares para descrever as regras de produção. Vamos definir a mesma gramática G utilizando a notação BNF.

```
<expr> ::= <expr> + <expr> | <expr> - <expr> | <expr> / <expr>
         | <expr> * <expr> | (<expr>) | <num>
<num> ::=  <dig> | <num><dig>
<dig> ::= "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"0"
```

A definição de ULLMAN, AHO & SETHI (1995) deixa mais clara o significado e a função da gramática nesta pesquisa. As regras de uma linguagem fonte são expressas através de gramática, onde são registradas todas as construções possíveis para esta linguagem. Um compilador utiliza em sua fase de análise as definições feitas na gramática da linguagem fonte para montar a árvore sintática. A árvore sintática serve de insumo para diversas outras operações executadas por um compilador.

ULLMAN, AHO & SETHI (1995) definem pontos importantes a respeito de gramática para compiladores:

- Gramáticas oferecem especificações sintáticas precisas e fácil de entender.

- A sintaxe das construções de uma linguagem de programação pode ser descrita utilizando gramáticas livre de contexto ou pela notação BNF.
- Para certas classes de gramáticas, pode-se construir automaticamente um analisador sintático (parser) que determine se um programa fonte está bem formatado.
- Uma gramática propriamente projetada implica na estrutura de linguagem de programação útil à tradução correta de programa-fonte em códigos-objetos e também na detecção de erros. Existem ferramentas que permitem converter a definição de gramática em programas operativos para leitura do código fonte.

Gramáticas livre de contexto geram linguagens livre de contexto. Uma linguagem é considerada “livre do contexto” quando suas regras de produções podem ser aplicadas independentemente do contexto do símbolo não terminal. (Hopcroft, Motwani, & Ullman, 2001).

2.2.1.1.1 Expressão regular

“Expressão regular é um formalismo denotacional, também considerado gerador, pois se pode inferir como construir (gerar) as palavras de uma linguagem. Uma expressão regular é definida a partir de conjuntos (linguagens) básicos e operações de concatenação e de união.” (Menezes, 2011).

Expressões regulares possibilitam definir regras de construção de uma linguagem e validar se sequências de caracteres pertencem a uma determinada linguagem. Elementos terminais de uma gramática podem ser definidos através de expressões regulares. Em muitos casos as expressões regulares são utilizadas em conjunto com notações gramaticais como a BNF.

2.2.1.2 Análise léxica

Em um compilador a fase de análise linear é chamada de análise léxica. Sua função básica é encontrar as sequências de menor significado na linguagem fonte chamados de *tokens* léxicos. Um *token* léxico é uma sequência de caractere contendo um significado e válido de acordo com a gramática da linguagem (Parr, 2009).

A análise léxica é a primeira etapa de um compilador. O trabalho executado nesta fase serve de entrada para a próxima fase, a análise sintática. O analisador léxico possui a função primordial de extrair os *tokens* descritos em linguagem fonte e transformá-los em uma lista de *tokens* que farão parte da árvore sintática que será construída pelo analisador sintático. Caso seja encontrado um *token* não definido na gramática da linguagem o analisador léxico gera erro no processamento do código fonte e o processo de compilação geralmente é interrompido (Ullman et al., 1995).

2.2.1.3 Análise sintática

A segunda fase de um compilador é a análise sintática. Neste passo é feita uma análise hierárquica chamada também de análise gramatical. Envolve o agrupamento dos *tokens* do programa fonte e frases gramaticais, que são usadas pelo compilador. (Ullman et al., 1995).

As regras de produção da gramática da linguagem fonte é aplicada pelo analisador sintático, o componente responsável pela análise sintática em um compilador. A medida que as regras de produção vão sendo derivadas o analisador sintático vai agrupando os símbolos de forma hierárquica, formato de árvore, tendo apenas *tokens* no nível folha. Dada uma gramática G definida conforme descrito a seguir:

```
<expr> ::= <expr> + <expr> | <num>

<num> ::=  <dig> | <num><dig>

<dig> ::= "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"0"
```

Ao processar uma expressão “2 + 1” gera-se a árvore exibida na Figura 3.

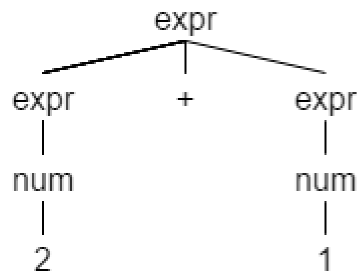

Figura 3 - Árvore sintática da gramática G expressão 2 + 1

ULLMAN, AHO & SETHI (1995) definem a função do analisador sintático como uma parte do compilador que irá obter uma cadeia de *tokens* (gerados pelo analisador léxico) e verificar se a mesma pode ser gerada pela gramática da linguagem fonte. Neste processo o analisador sintático relata quais são as inconformidades da combinação de *tokens* com as definições feitas na gramática da linguagem. O analisador sintático ainda deve se recuperar de qualquer erro sintático encontrado a fim de detectar outros possíveis erros na lista de *tokens*.

O analisador sintático além de gerar a árvore sintática, que servirá de insumo para as demais fases do compilador, analisa se as construções feitas no programa fonte são condizentes com a gramática que define a linguagem fonte. Essa validação garante que qualquer instrução feita pertence realmente a linguagem fonte e que as demais fases do compilador, como a análise semântica poderão analisar todas as construções encontradas.

A árvore sintática, produto da análise sintática, permite a criação mais simplificada de programas que navegam pelo código fonte. Apenas a árvore sintática não é suficiente para o completo entendimento do código fonte. Na seção Análise Semântica, poderá ser observado que a árvore sintática será acrescida de informações que permitirão o pleno entendimento das construções de um programa.

Utilizando a árvore sintática já é possível fazer cálculos de complexidade do código, localização de padrões de codificação de programadores (limitados ao escopo local do código fonte). Também é possível percorrer a árvore sintática para fins de tradução da linguagem fonte para uma linguagem alvo, porém sem informações semânticas a tradução fica limitada e as linguagens precisam estar em níveis próximos com tipo de dados similares (Parr, 2009).

2.2.1.4 Análise Semântica

A fase de análise semântica verifica os erros semânticos no programa fonte e captura as informações de tipo para a fase subsequente que é a geração de código. É feita leitura da árvore sintática, a fim de identificar o operador e operando das expressões e enunciados (Ullman et al., 1995). Uma parte importante da análise semântica é a verificação de tipos, onde o compilador pode analisar qual o tipo dos elementos envolvidos em uma operação e validar se os mesmos são compatíveis. Também é função do analisador semântico fazer elevação de tipos, que ao avaliar uma operação entre dois tipos distintos o resultado da expressão é do tipo de maior precisão, ou o tipo definido pelos padrões da linguagem fonte. Como exemplo temos a multiplicação de dois números, um inteiro e outro real. O tipo do resultado desta multiplicação na maioria das linguagens seria real. As informações de tipo são muito relevantes para o compilador não apenas para validação de compatibilidade entre tipos e operações, mas, em linguagens mais recentes a inferência de tipos é utilizada cada vez com mais frequência, como exemplo temos a sobrecarga de métodos na Programação Orientada à Objetos (POO) que depende do tipo dos argumentos para determinar qual trecho de código executar em seguida (Parr, 2009).

O analisador semântico varre a árvore sintática localizando símbolos e os catalogando. A medida que os símbolos são encontrados eles vão sendo armazenados na tabela de símbolos conforme veremos na seção Tabela de símbolos. Os símbolos são catalogados e classificados. Dependendo da linguagem fonte a busca por símbolos precisa ser feitas de vários passos.

Primeiro o analisador semântico procura por símbolos que definem tipos, como em linguagens orientadas a objetos onde classes são definições de tipo, em uma seguida procura pelos símbolos que compõem os tipos como métodos e atributos e em uma terceira passagem catalogar os símbolos do código.

2.2.1.4.1 Verificação de tipos

Verificação de tipos é um processo de verificar a coerência de tipo do programa fonte usando regras lógicas para verificar o comportamento de um programa em tempo de compilação ou em tempo de execução. Ele permite que os programadores limitem os tipos que podem ser usados para aspectos semânticos de compilação. Ele atribui tipos à valores e também verifica se estes valores são coerentes com sua definição no programa.

Verificação de tipo também pode ser usada para a detecção de erros em programas. Mas erros podem ser verificados dinamicamente (em tempo de execução) se o programa alvo contém o tipo de um elemento e seu valor, mas um sistema de tipos em tempo de compilação elimina a necessidade de verificação dinâmica de erros, garantindo que estes erros não surgirão quando o programa for executado. Se as regras de verificação de tipo são aplicadas fortemente (isto é, permitindo que apenas as conversões automáticas de tipo, que não resultam em perda de informações), então a implementação da linguagem é dita ser fortemente tipada; caso contrário, diz ser fracamente tipada. Uma implementação da linguagem com rigidez garante que o programa de destino será executado sem qualquer erro de tipos.

2.2.1.4.2 Sistema de tipos

Um sistema de tipos é uma coleção de regras para expressões de tipos das várias partes de um programa. Um verificador de tipos implementa um sistema de tipos.

O sistema de tipos permite que o sistema de verificação de tipos valide a compatibilidade entre tipos, permite também calcular o resultado final de uma expressão. O tipo da expressão

"x * y" só pode ser definido se soubermos o tipo das variáveis x e y. Caso ambas sejam do tipo inteiro o resultado da expressão é do tipo inteiro, para essa conclusão não foi necessário o uso do sistema de tipos. Caso x seja real e y seja um inteiro o tipo do resultado só pode ser obtido através do sistema de tipos.

O sistema de tipos é divido em dois grupos: sistema de tipos básicos e sistema de tipos construídos. O sistema de tipos básicos contém os tipos atômicos, também denominados primitivos como inteiros, booleanos, reais. Os tipos construídos são os arrays (vetores), estruturas, classes e outros.

O sistema de tipo ainda é responsável por validar se tipos são iguais ou equivalentes. Esta tarefa se torna mais árdua quando envolvemos linguagens que utilizam da POO, pois a complexidade de tipo aumenta quando envolve herança entre tipos. O sistema de tipo deve verificar se um determinado tipo é compatível com o outro, determinando assim se valores podem ser atribuídos, como no caso de variáveis ou passagem de parâmetros.

2.2.1.4.3 Tabela de símbolos

Uma tabela de símbolos é uma estrutura de dados de tempo de compilação que é usada pelo compilador para coletar e utilizar informações sobre as construções de programa de fonte, como variáveis, constantes, funções, etc. Tabela de símbolos ajuda o compilador na determinação e verificação da semântica de determinado programa fonte. As informações na tabela de símbolos podem ser inseridas nas fases de análise léxica e análise sintática, no entanto, é usado nas fases posteriores do compilador (análise da semântica, geração de código intermediário, otimização de código e geração de código). Intuitivamente, uma tabela de símbolo mapeia nomes em declarações (chamadas de atributos), por exemplo, mapear um nome de variável ao seu tipo de dados "char".

Cada vez que um nome é encontrado no programa fonte, o compilador procura na tabela de símbolos. Se o compilador encontra um novo nome ou novas informações sobre um nome

existente, ele modifica a tabela de símbolos. Assim, deve prever-se um mecanismo eficiente para recuperar as informações armazenadas na tabela, bem como para adicionar novas informações para a tabela. As entradas na tabela de símbolos são compostas pelo par de chave valor (nome e informações). Por exemplo, para a seguinte instrução de declaração de variável "string a" gera uma entrada na tabela de símbolos contendo o nome da variável "a", como chave, e "string" como uma informação sobre a variável (tipo).

Uma tabela de símbolos deve conter as seguintes funcionalidades:

- Lookup – determinar se um símbolo já existe na tabela
- Insert – inserir novas entradas de símbolo
- Access – acessar informações de um determinado símbolo
- Modify – modificar as informações de um determinado símbolo
- Delete – excluir um símbolo ou grupo de símbolos da tabela

Outras preocupações que se deve ter com a tabela de símbolo e os componentes do compilador é em relação a símbolos que possuem o mesmo nome. Algumas linguagens permitem que tipos diferentes de símbolos possuam o mesmo nome outras permitem até que símbolos do mesmo tipo possuam nomes iguais, como métodos sobrecarregados na POO.

2.2.1.4.4 Escopo de símbolos

O escopo define onde um determinado símbolo é válido. Linguagens diferentes têm diferentes sistemas de escopo para declarações. Por exemplo, em FORTRAN, o escopo de um nome é uma sub-rotina única, enquanto em ALGOL, o escopo de um nome é a seção ou o procedimento no qual é declarada. Em linguagens mais modernas o escopo pode ser definido em um pequeno bloco como loop de repetição. Assim, o mesmo identificador pode ser declarado várias vezes, com diferentes atributos e locais de armazenamento. A tabela de símbolos, portanto, é responsável por manter as declarações do mesmo identificador distintas.

Para fazer a distinção entre as declarações, um número exclusivo é atribuído a cada elemento de programa identificando seu escopo e seu identificador único (normalmente uma referência para o elemento na árvore sintática).

Cada linguagem tem a sua particularidade de escopo que podem ser global, instância de um tipo como classe ou estrutura, local de um procedimento ou até mesmo local de um bloco de código com um loop de repetição. É possível ter elemento distinto, porém com o mesmo nome em cada um destes níveis de escopo, e o analisador semântico deve saber exatamente a qual um trecho do código fonte está se referindo.

Os escopos se relacionam de forma hierárquica. A medida que um elemento é encontrado ele é registrado na tabela de símbolos associado ao escopo mais próximo. Quando o analisador semântico precisa de informações de um elemento específico ele recorre a tabela de símbolo no escopo corrente. Caso não encontre informação o escopo anterior é consultado e assim sucessivamente até que o símbolo seja encontrado ou o analisador semântico gere um erro pois o elemento não foi encontrado em nenhum dos escopos aninhados.

2.2.2 Geração do programa alvo

Após as análises sintática e semântica, alguns compiladores geram uma representação intermediária explícita do programa fonte. Essa representação intermediária deve possuir duas propriedades importantes: ser fácil de produzir e fácil de traduzir no programa alvo (Ullman et al., 1995).

A representação intermediária pode ter uma variedade de formas. A flexibilidade de transformar uma linguagem fonte em uma linguagem intermediaria facilita o processo de tradução. A linguagem intermediária permite dividir a primeira parte do compilador, a análise da linguagem fonte, da segunda parte geração do programa alvo. Uma vez definida uma linguagem intermediária o processo de compilação pode ser desacoplado para que vários

analisadores de linguagem fonte diferentes possam gerar a mesma linguagem intermediaria e serem escritas na linguagem alvo pelo mesmo gerador de código. Outra abordagem pode ser que uma mesma linguagem alvo pode ser gerada a partir de várias linguagens fonte.

Normalmente o projeto da linguagem intermediária é próxima à da linguagem de máquina, uma vez que a maioria dos compiladores possuem como linguagem alvo a linguagem de máquina para gerar programas executáveis. (Ullman et al., 1995) definem um padrão de três endereços bem próximo da linguagem *assembly*, porém algumas abordagens inclusive a abordada nesta pesquisa prevê a compilação de uma linguagem de alto nível para outra de mesmo nível ou superior este modelo também é detalhado por (K.V.N. Sunitha, 2013).

No caso de compilação entre linguagens de alto nível a linguagem intermediária deve ser também de alto nível e extremamente dotada de notações semânticas. A tradução entre linguagens de alto nível pode esbarrar em problemas de recursos ou expressões idiomáticas não presentes na linguagem alvo, ou muita diferença entre expressões idiomáticas equivalentes (K.V.N. Sunitha, 2013).

2.3 Tamanho de sistemas

Um método de estimativa é um conjunto de processos, suportados por fórmulas empíricas adequadas e apoiada por dados de referência histórica, que ajudam a derivar resultados previsíveis dentro de um nível decente de precisão. Há um número de métodos de estimativa de projeto de software disponíveis para gerentes de projetos que ajudam a chegar ao tamanho estimado, esforço, agenda, carregamento de recursos e outros parâmetros semelhantes (Parthasarathy, 2007).

Métodos de estimativa também podem ser utilizados para medir. Existe uma diferença a respeito de estimar e medir a figura 4 demonstra essa diferença.

Estimativa: Cálculo de predição do tamanho do sistema. Executado antes da construção do sistema.

Medição: Calculo e aferição, para medir o tamanho do sistema já pronto. Com a medição é possível comparar dois sistemas já existente para medir o tamanho de ambos.

Já a complexidade determina não o tamanho, mas quão complexo é um sistema.

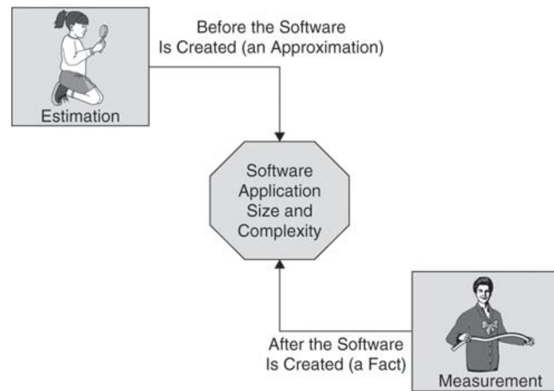

Figura 4 - Estimativa e medição –

Fonte: (Parthasarathy, 2007)

2.3.1 Tamanho funcional

Tamanho funcional é uma medida de tamanho de software, baseada em uma avaliação padronizada dos requisitos lógicos dos usuários. Na indústria há atualmente várias maneiras para se medir tamanho funcional, a mais antiga das quais são os pontos de função. Semelhante aos metros quadrados de uma casa, pontos de função são independentes dos métodos físicos, ferramentas ou linguagem de desenvolvimento utilizados para construir o software. Outras medidas que alegam também medir o tamanho funcional do software incluem Pontos de Função Mark II, Pontos de Função 3D da Boeing e Pontos de Característica (Dekkers, 1999).

2.3.1.1 Pontos de função

Quase todas as disciplinas de engenharia dependem da premissa básica de ser capaz de medir os vários parâmetros com os quais lida esse ramo específico de engenharia. Por exemplo, engenheiros civis podem medir várias estruturas usando o sistema métrico de medição; engenheiros elétricos podem medir através de unidades como watts, volts, amperes, etc.; e engenheiros mecânicos medem saídas através de joules, cavalos de potência e outras escalas de

medição. Mas a engenharia de software é relativamente jovem e ainda estamos para chegar à unidade de medida para todos os sistemas de software bem definida e universalmente aceitável. Os pontos de função são o mais próximo de se tornar uma unidade universal de medição de sistemas de software, baseado sobre as funções de negócio entregadas pelo aplicativo. Linhas-de-código é uma alternativa muito pobre (Parthasarathy, 2007). Um ponto de função é uma unidade de medida que permite medir tamanho de um software. O software medido é proporcionalmente igual a entrega funcional deste aplicativo ao usuário (Dekkers, 1999). Assim através da Análise de Ponto de Função (FPA) é possível comparar dois sistemas em quanto ao seu tamanho funcional.

2.4 Tecnologias e ferramentas

2.4.1 A linguagem C#

C# é uma linguagem de programação simples, moderna, orientada a objetos e de tipo seguro. C# tem suas raízes na família da linguagem C e é familiar aos programadores C, C++ e Java. C# é padronizado pela ECMA International como padrão ECMA-334 e pela ISO/IEC como norma ISO/IEC 23270 (Hejlsberg, Golde, Wiltamuth, & Torgersen, 2010). Criada por Anders Hejlsberg, também criador do TurboPascal e arquiteto do Borland Delphi possui seu nome inspirado na linguagem em que se baseia, a linguagem C. Originalmente chamada de Cool (*C like Object Oriented Language*) o nome foi alterado para C# (lê-se cê charp) dando alusão a C++++ em relação ao C++ e a nota dó sustenido representada pelo mesmo símbolo (C#) que é um semitom acima da nota dó (C) (Hamilton, 2008).

C# é uma evolução das linguagens C e C++ implementadas pela Microsoft. Diversos conceitos foram implementados como *Garbage Collector*, referências em detrimento à ponteiros entre outras (MSDN, 2003). Anunciada pela Microsoft em 2000 o C# foi introduzido em 2002 no .Net Framework e está em constante evolução atualmente em sua versão 7.0 lançada em março de 2017.

O C# é uma linguagem de alto nível orientada a objeto e possui acesso ao poderoso *.Net Framework Class Library,* uma vasta coleção pré-existente de classes que habilitam o desenvolvedor a criar aplicações rapidamente. Possui diversas funcionalidades como: portabilidade para diversas plataformas, frameworks para sistemas Web, recursos para programação assíncrona. O C# foi projetado para rodar sobre a plataforma Microsoft.Net com o .Net Framework (Deitel & Deitel, 2016).

2.4.1.1 Microsoft .Net

Em 2000 a Microsoft anunciou o lançamento da plataforma .Net com o .Net Framework 1.0. Era uma nova plataforma onde aplicações robustas tanto para internet quanto para desktop poderiam ser criadas (Deitel & Deitel, 2016). O .Net Framework era até então a infraestrutura para executar programas da plataforma .Net com o JIT[1] *compiler* responsável por garantir que o código seja executado na plataforma alvo (Engel, 1999). Com esta proposta o .Net Framework possibilita a portabilidade dos programas para diversas plataformas diferentes.

O .Net Framework é o nome comercial utilizado para especificar o *Common Language Infrastucture* (CLI). O CLI foi desenvolvido pela Microsoft e padronizado pela ISO e pela ECMA e tem evoluído constantemente (Ky, 2016). A plataforma .Net foi trifurcada devido ao porte para diversas plataformas como pode ser visto na figura 5:

- A versão clássica que mantém a mesma base e compatibilidade com as primeiras versões do .Net
- A .Net Core iniciativa para criação de uma plataforma mais portável para outras plataformas que não o Microsoft Windows

[1] Just In Time compiler – compiladores responsáveis por transformar código intermediário em código de máquina em tempo de execução de um programa.

- A Mono versão clássica portada para outras plataformas que não Windows hoje fortemente ligada a tecnologias móveis.

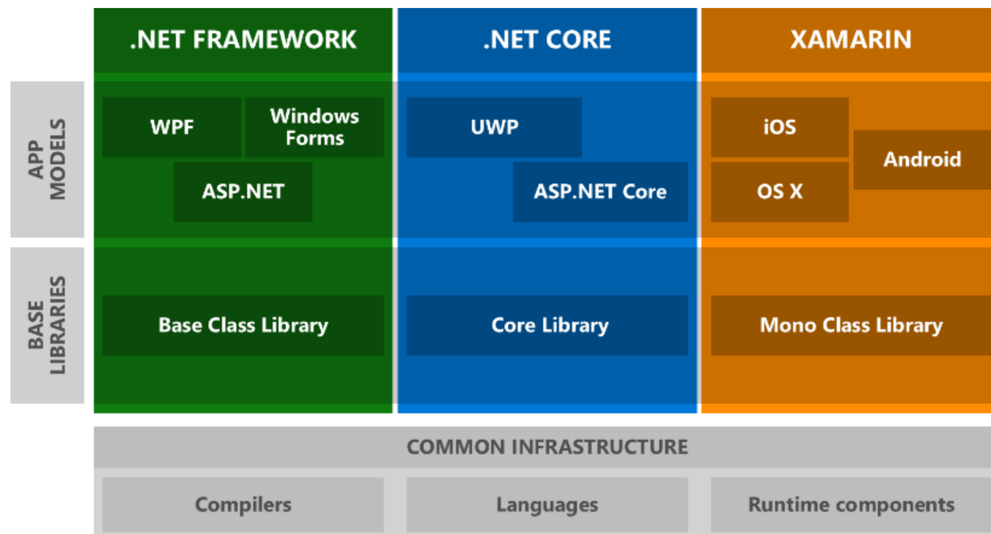

Figura 5 – Trifurcação do .Net Framework

Fonte: (Landwerth, 2016)

Porém atualmente as três vertentes da plataforma foram unificadas no *.Net Standard*. O .Net Standard é um padrão que prevê a unificação das bibliotecas pré-definidas nas três vertentes do .Net possibilitando o compartilhamento de código dos programas multi-plataformas (Landwerth, 2016). A figura 6 demonstra como funciona o .Net Standard.

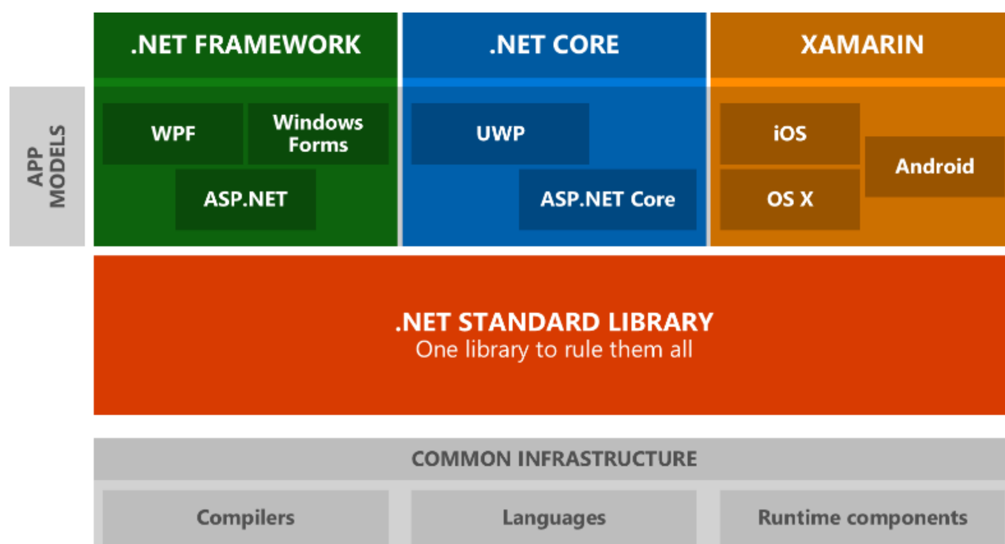

Figura 6 - .Net Standard

Fonte: (Landwerth, 2016)

2.4.2 A Plataforma PowerBuilder

PowerBuilder é um dos primeiros exemplos de *Integrated Development Envirionment* (IDE). Este conceito introduziu velocidade no desenvolvimento de sistemas uma vez que não era mais necessário sair do ambiente de desenvolvimento para realizar tarefas relacionadas à ferramenta de desenvolvimento (Brown & Armstrong, 2003).

Criado incialmente pela Powersoft que foi adquirida pela Sybase em 1995 que posteriormente foi adquirida pela SAP em 2010 que por sua vez, em 2016, delegou a manutenção e comercialização do PowerBuilder para a Appeon que atualmente mantem a ferramenta (Berner, 2016). O PowerBuilder era uma das ferramentas mais populares de desenvolvimento na década de 90 (Bianco, 2010). E hoje tem perdido espaço no mercado apesar dos esforços da Appeon, sua detentora (Appeon, 2017; Brandel, 2007).

Um código fonte PowerBuilder pode conter basicamente dois elementos distintos, DataWindows® (DW) e código PowerScript® que são os códigos que contém a lógica do programa. As DWs são objetos do PowerBuilder que permitem recuperar, apresentar e manipular dados de um banco de dados relacional ou qualquer outra fonte de dados (Sybase, 2011, 2012b).

2.4.3 O Compilador Roslyn

O Roslyn é o novo compilador da Microsoft. Ele não representa apenas uma reescrita dos compiladores pré-existentes, mas uma repaginação do processo de compilação em geral. Antigamente os compiladores das linguagens da plataforma .Net eram uma caixa preta. Linhas de código entravam e um assembly compilado saía, era impossível interagir com o processo de compilação acessando os passos intermediários da compilação. Essa abordagem foi completamente modificada no Roslyn que agora disponibiliza uma API que permite criar

programas que interajam com o compilador (Nick Harrison, 2017). A figura 7 demonstra os grupos de APIs disponíveis no compilador e que podem ser consumidas por qualquer outro programa.

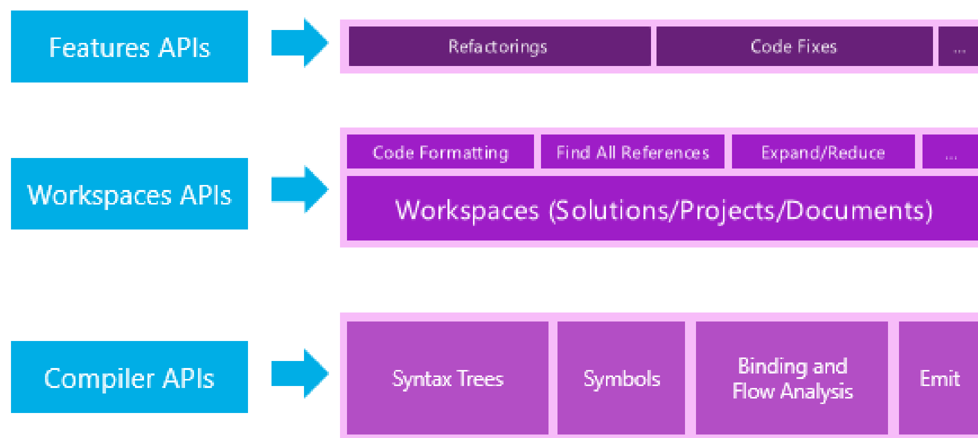

Figura 7- Estrutura das APIs do Roslyn

Fonte: (Parsons, 2015)

As APIs do compilador são divididas em duas camadas a *Compiler API* e a *Workspace API*. A *Compiler API* possui as funções de compilação do Roslyn como: análise léxica e sintática, análise semântica, análise de fluxo de execução e a emissão do código final que é a geração do código C# ou VB.Net compilado para MSIL linguagem intermediária da Microsoft posteriormente compilada para linguagem de máquina pelo JIT do .Net Framework. A *Workspace API* é focada no código fonte e possui funções que permitem a manipulação de código *source-to-source*. Com a *Workspace API* é possível navegar por fonte C# ou VB.Net fazendo analise e modificações no fonte. Isso só é possível devido a SyntaxFactory que permite gerar códigos em C#. Outra função da *Workspace API* é a formatação do fonte de acordo com os padrões e conversões da linguagem (Parsons, 2015).

2.4.4 ANTLR

ANTLR (*ANother Tool for Language Recognition*) é um poderoso gerador de analisadores léxico sintáticos ou *parser*. Ele gera *parsers* que utilizam a estratégia de analise

LL(*) que faz a leitura da gramatica como demonstrado na seção Análise do programa fonte (Parr & Fisher, 2011).

O ANTLR permite através da escrita de uma gramatica baseada no padrão BNF criação de *parsers* que geram ASTs (*Abstract Syntax Tree*) permitindo a navegação na estrutura processada. O trabalho do ANTLR é transformar a gramática em código na linguagem desejada, como C# e Java. O código gerado representa as construções conforme a teoria de autômatos finitos (Parr, 2009).

O ANTLR foi escolhido pelo pesquisador por afinidade com a linguagem utilizada para definir gramáticas e pelo fato de gerar código (para reconhecimento de construções idiomáticas) na linguagem C#, que também é de domínio deste pesquisador.

2.5 Trabalhos relacionados

Na literatura existem vários exemplos de construção de processos automatizados de migração de código entre linguagens de desenvolvimento. Estes trabalhos serão utilizados como base para o desenvolvimento desta pesquisa. Os desafios encontrados durante o processo serão replicados nesta pesquisa e catalogados. Nesta seção serão apresentados trabalhos sobre automação de migração de código fonte e a relação destes com esta pesquisa.

(Verhoef & Terekhov, 2000) descrevem neste trabalho os desafios da migração de código entre linguagens. É feito um panorama das diferenças de construção de várias linguagens legadas para linguagens recentes e compila as principais dificuldades da migração de código, seja ela automática ou manual. O trabalho ainda lista uma série de requisitos que um processo automatizado de migração de código fonte deveria ter. Vários desafios listados por este trabalho serão utilizados nesta pesquisa.

(Martin & Muller, 2001) descrevem diversas estratégias para integrar código fonte feito em linguagem C com programas executando na plataforma Java. Ele demostra diversas estratégias para justificar que a conversão automatizada do código fonte de C para Java é a

melhor estratégia. Neste trabalho é desenvolvida uma ferramenta chamada Ephedra que converte código C em Java e relata todo o processo utilizado para transformar as construções. O trabalho se limita à linguagem C que não é orientada a objetos e não possui alguns dos desafios que serão enfrentados nesta pesquisa.

(Terekhov, 2001) descreve a análise da utilização de uma ferramenta automatizada para migrar de uma linguagem proprietária chamada Rules para COBOL e VB. O trabalho não demonstra os passos para a criação da ferramenta de conversão, mas expões as dificuldades de migração de Rules para COBOL e para VB.

(Mossienko, 2003) descreve o processo automatizado para migração de COBOL para Java. Neste trabalho ele descreve os procedimentos mais desafiadores, porém não descreve como o programa de tradução foi feito. O processo não envolve apenas a transformação, mas também um aprimoramento do código gerado, fazendo com que ele seja melhor que o fonte original em termos de legibilidade, este passo é feito manualmente.

(El-Ramly et al., 2006) descreve a criação de uma ferramenta chamada Java2C# que converte programas que tem o Java como linguagem origem para C#. A ferramenta é escrita utilizando TXL onde se escreve a gramática da linguagem origem e as regras de transformação para a linguagem alvo. Este trabalho se assemelha bem ao desta pesquisa pois além de estruturar um compilador para transformação de código as transformações são classificadas em relação ao tamanho do desafio para implementá-la. Neste trabalho algumas transformações com nível de desafio muito alto são marcadas para intervenção manual, ou seja, não são migradas, o que difere desta pesquisa pois o critério de migração ou não de uma funcionalidade estará ligada à sua frequência nos fontes existes. Não existe análise sobre esforço manual dos itens não migrados neste trabalho.

(Kontogiannis et al., 2010) descreve a criação de um conversor de PL/I para C++. Neste trabalho ele relata, assim como se espera dessa pesquisa, o processo de criação da automação

da conversão de código expondo técnicas utilizadas para mapeamento do código fonte origem para o código fonte alvo. Apesar de expor soluções de transformação de código a linguagem do programa de origem, PL/I, não possui todos os elementos da linguagem origem desta pesquisa por ser uma linguagem mais simples. O PowerBuilder utiliza uma linguagem mais recente e com mais recursos que o PL/I, assim tendo mais desafios para sua migração. Outro ponto em comum com esta pesquisa é que o processo automatizado não se propõe a migrar 100% do código, porém o artigo não propõe modelo para avaliar o custo da intervenção manual.

3 METODOLOGIA

3.1 Considerações iniciais

Esta é uma pesquisa exploratória com objetivo de aumentar o conhecimento em relação aos processos automatizados de modernização de software identificando seus desafios. Esta se baseia na automação da migração de código fonte entre plataformas diferentes. As etapas estão norteadas pelos objetivos citados na seção 1.2 Objetivos. A figura 8 demonstra o rastreamento dos objetivos desta pesquisa. Cada objetivo está sendo atendido por alguma etapa da pesquisa.

Etapa 1 – Construção do Compilador	Etapa 2 – Conversão Automatizada	Etapa 3 – Workshop	Etapa 4 – Síntese dos Desafios	Etapa 5 – Análise de Viabilidade
• Obj. Esp. 1	• Obj. Esp. 3 • Obj. Esp. 4 • Obj. Esp. 5	• Obj. Esp. 3 • Obj. Esp. 4	• Obj. Esp. 2	• Obj. Esp. 5

Figura 8- Etapas x Objetivos

3.2 Etapas da Pesquisa

A pesquisa foi executada em uma empresa de desenvolvimento de software cujo os sistemas legados estão sobre a plataforma PowerBuilder. Por este motivo a escolha desta como linguagem/plataforma origem. A empresa possui expertise em desenvolvimento de softwares na plataforma .net utilizando a linguagem C# e este o motivo da escolha desta linguagem como linguagem alvo.

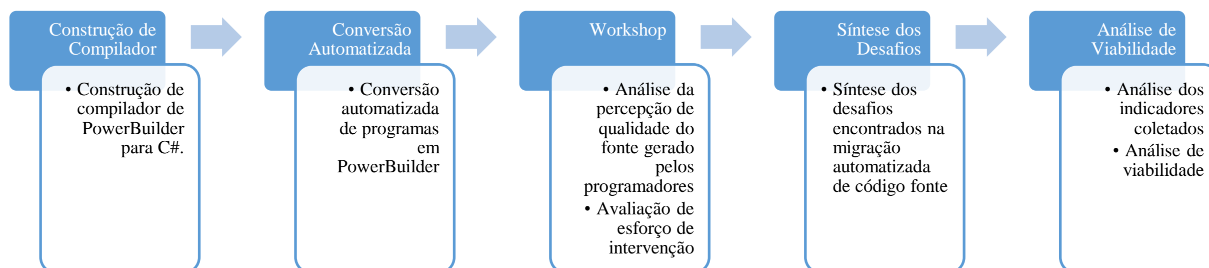

Figura 9 - Etapas do projeto

A pesquisa foi executada em cinco etapas que estão descritas na figura 9. A primeira etapa foi responsável por criar o compilador. A segunda etapa corresponde a execução da migração (conversão) automatizada de programas PowerBuilder para C# onde foram selecionados dois programas para a migração. Na terceira etapa um workshop foi realizado

avaliando através de questionário a qualidade do código fonte gerado e através de reuniões coletivas foi obtido uma estimativa de esforço (homem/hora) para finalizar os programas manualmente, uma vez que o compilador não converteu 100% do código. Na quarta etapa foi gerada uma síntese dos desafios da migração do código, onde são apresentados os principais complicadores do processo e como foram superados. Na quinta e última etapa foram realizadas análises financeiras demostrando a viabilidade econômica do processo.

3.3 Construção do compilador

Em todo processo de conversão espera-se que o programa alvo se comporte da mesma forma que o programa fonte. O processo de conversão manual é custoso e pode conter erros provocados por falha humana (PIRKELBAUER, 2010). A migração manual ainda pode ser indicada para pequenas quantidades de código fonte. Para a migração de grandes quantidades de fonte um processo automatizado é o mais indicado e requer um compilador (Chisolm & Lisonbee, 1999).

A construção do compilador foi dividida em duas fases: a análise e a geração. A construção deste compilador atendeu ao objetivo específico 1.

3.3.1 Análise

A fase de análise é responsável por ler o programa fonte entendendo as construções existentes no mesmo.

Devido à característica da plataforma dos programas de origem, o PowerBuilder, foi necessário criar um processo adicional no compilador responsável por extrair o código fonte de algumas estruturas proprietárias da plataforma.

A análise léxica e a análise sintática foram feitas utilizando a ferramenta ANTLR. Uma gramática descrevendo a linguagem PowerScript, objetos DataWindow e a linguagem SQL (com adaptações do PowerScript) foram escritas, em linguagem definida pela ANTLR. Os

processadores gerados pelo ANTLR foram utilizados para a geração da Arvore de Sintaxe Abstrata (AST) que permitie a navegação pelo código fonte o que foi é crucial para o funcionamento do compilador.

Devido à natureza das linguagens envolvidas neste experimento, que se pode extender á maioria das linguagens modernas, se fez necessário a criação de um analisador semântico. O analisador semantico neste processo irá auxiliar a geração de códigos mais legíveis e aumentar o percentual de linhas de código convertidas sem erro. Segundo (Verhoef & Terekhov, 2000) o maior problema de conversores de código automáticos é a diferenciação de tipos entre as plataformas e o analisador semântico junto com seu sistema de tipos será crucial neste processo. Na etapa Síntese dos desafios fica evidente a necessidade do analisador semântico, uma vez que a maioria das soluções dos principais desafios dependem de informações geradas pelo analisador semântico.

3.3.2 Geração

O objetivo final de todo compilador é a geração do código na linguagem alvo, seja ela de alto nível como o C#, PowerBuilder, Java, ou de baixo nível como: linguagem de máquina, assembly, bytecode e MSIL[2].

Nesta pesquisa o código é gerado em C# que ao contrário do PowerBuilder possui ferramentas para análise e transformação de código disponíveis para uso. Em 2015 o C# teve seu compilador totalmente reescrito introduzindo novos conceitos como prestação de serviço. Este novo compilador é chamado de *Roslyn* e traz um conceito de um “compilador como serviço” onde diversas APIs estão disponíveis para serem consumidas pelos programas (Parsons, 2015).

[2] Bytecode (java) e MSIL(.Net) são linguagens intermediárias aos quais códigos fonte em java e .Net, respectivamente, são compilados

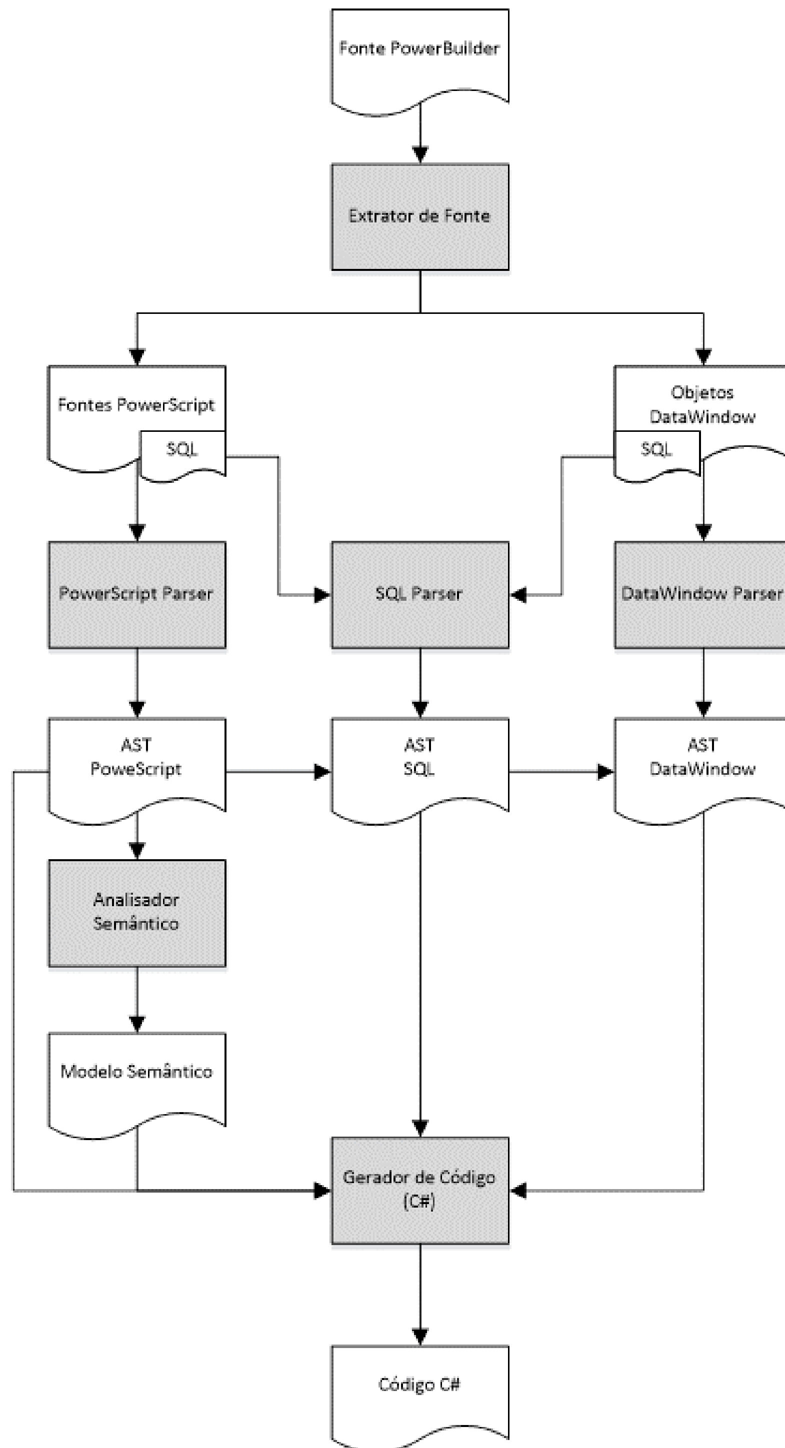

Figura 10 - Fluxo de conversão do código fonte PowerBuilder para C#

Uma das APIs do *Roslyn* é a *Workspace API* que possui a função de geração e formatação de código. Esta API foi utilizada para a geração de códigos em C# nesta pesquisa.

A geração do programa alvo utiliza como entrada, como descrito na figura 10, a AST e o modelo semântico do programa fonte. As ASTs de PowerScript criadas pelo analisador

léxico-sintático são percorridas de forma recursiva por cada elemento do código de forma gerar as estruturas correspondentes na linguagem alvo.

Com intuito de restringir o escopo e reduzir o custo de implementação do compilador, não serão feitas conversões de interfaces gráficas. Migração deste tipo de programa são tratados como estudos futuros neste trabalho e foram devidamente citados na conclusão deste trabalho.

3.3.3 As etapas do compilador

O compilador foi construído em cinco etapas: o extrator, o analisador léxico-sintático, analisador semântico, uma biblioteca de componentes e o gerador de código alvo. Nesta seção será detalhada a criação do compilador etapa por etapa de forma a permitir que este procedimento seja repetido em outras pesquisas.

O compilador funciona no fluxo descrito na figura 11 onde cada etapa gera artefatos para a etapa seguinte. A ultima etapa a geração do código utiliza insumo das duas etapas antecessoras, pois ele faz uso do AST e da tabela de símbolos.

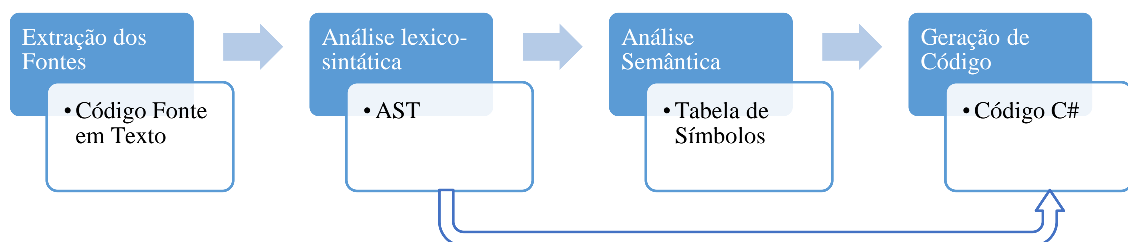

Figura 11 - Fluxo de Execução do Compilador

Transformações simples chamadas de transformações diretas são feitas utilizando apenas o AST já transformações mais complexas demandam acesso à tabela de símbolos. Mais detalhes sobre as transformações poderão ser vistos na seção Síntese dos desafios onde é detalhado o conceito das transformações.

A figura 11 não demostra a biblioteca de componentes, pois ela não faz parte do fluxo de execução do compilador, assim o compilador não depende da existência da biblioteca para ser executado, porém ela é fundamental para que o código gerado pelo compilador seja executado. As transformações feitas pelo compilador irão descrever construções existentes na biblioteca, assim o código gerado referencia a biblioteca e não o próprio compilador.

3.3.3.1 Extrator de fonte

O PowerBuilder mantém seus arquivos fonte dentro de um container chamado PBL (PowerBuilder Library). Este arquivo (PBL) é de formato proprietário e não foi encontrada documentação do formato do mesmo. Para que o código fonte possa ser analisado e convertido primeiro ele deve ser extraído do arquivo PBL.

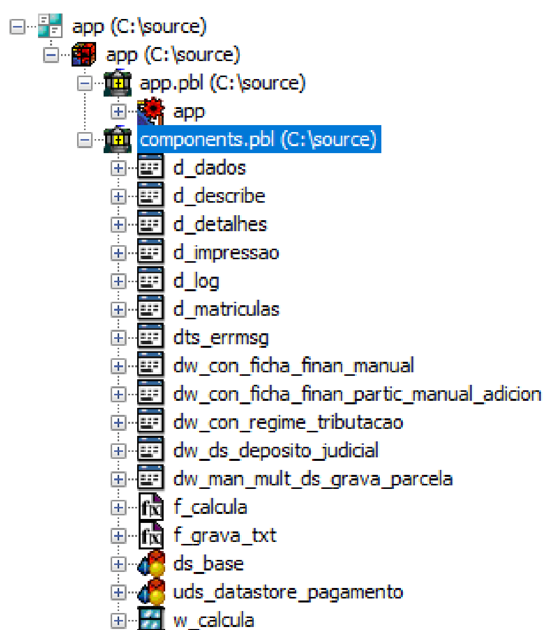

Figura 12 - Estrutura de código fonte PowerBuilder

O código fonte é estruturado em três níveis (*Workspace*, *Targets* e PBLs) que pode ser observado na figura 12. Os arquivos de *workspace* e *target* possuem formato texto e foi construído um *parser* simples para analisá-los e identificando os targets (no *workspace*) e as

PBLs (nos targets), porém o formato dos arquivos PBL não é conhecido e para isso foi necessário utilizar comandos do próprio PowerBuilder para localizar e extrair os arquivos fonte.

A plataforma conta com duas funções *LibraryDirectory* e *LibraryExport* (Sybase, 2012b) que permitem respectivamente listar os fontes contidos em uma PBL e exportá-los como arquivo texto.

Uma aplicação escrita em PowerScript (na própria plataforma PowerBuilder) foi projetada para prestar o serviço necessário de extração das fontes (em formato texto) para poderem ser analisados pelo compilador. Esta etapa não é muito usual nas demais plataformas, mas não descarta a necessidade desta etapa na aplicação dos modelos desta pesquisa em outros contextos.

3.3.3.2 Analisador léxico-sintático

Devido a inexistência de um analisador sintático para PowerBuilder que estivesse disponível para a pesquisa, foi necessário a criação de um *parser* dos elementos que compõem o código fonte PowerBuilder.

Um código fonte PowerBuilder pode conter basicamente dois elementos distintos, DataWindows® (DW) e código PowerScript® que são os códigos que contém a lógica do programa. As DWs são objetos do PowerBuilder que permitem recuperar, apresentar e manipular dados de um banco de dados relacional ou qualquer outra fonte de dados (Sybase, 2012a). Neste tipo de arquivo são extraídos principalmente os comandos em linguagem SQL para serem analisados. Os objetos DW possuem um formato bem específico e por isso um analisador sintático específico para estes objetos foi ser criado.

A linguagem PowerScript suporta comandos SQL (Structured Query Language) embutidos ou seja é possível no corpo do código fonte incluir comandos como SELECT, INSERT, UPDATE e DELETE (Sybase, 2012b). Para este recurso foi construído um analisador sintático específico, pois os comandos possuem pequenas variações dos comandos

padronizados (ISO, 2016) para melhor integração com o ambiente PowerBuilder e a linguagem PowerScript. Nesta pesquisa os programas fonte fazem acesso somente a base de dados gerenciadas pelos sistemas SQL Server[3] e Oracle[4], portanto este analisador suporta apenas os comandos definidos nas especificações destes dois sistemas gerenciadores de banco de dados.

Para analisar os códigos fonte de forma que estes possam ser convertidos no processo de geração de código alvo foi necessário a construção de três analisadores sintáticos.

O primeiro analisador faz análise dos elementos codificados na linguagem PowerScript seguindo o definido no manual da linguagem. O segundo faz a análise dos objetos DW seguindo o definido no guia de referência deste objeto. O terceiro faz análise dos comandos SQL de acordo com a referência das linguagens T-SQL[5] e PL-SQL[6].

Para a construção destes analisadores foi escolhida ferramenta ANTLR versão 4 devido a facilidade de descrever a gramática nesta ferramenta/versão. Foram construídas as gramáticas conforme especificado na referência das linguagens envolvidas e gerado, através da ferramenta: o processador léxico e o processador sintático (parser) em C#. O ANTLR poderia gerar os processadores em outras linguagens como Java e C++. Devido a afinidade deste pesquisador com a linguagem C# está foi utilizada.

Os processadores gerados pelo ANTLR são utilizados para a geração da Arvore de Sintaxe Abstrata (AST) que permite a navegação pelo código fonte. Conforme descrito na figura 10 os ASTs gerados pelos analisadores sintáticos de PowerScript, SQL e DW são consumidos em etapas posteriores do compilador. O analisador semântico e o gerador de código C# fazem uso destes produtos para a geração do programa alvo. Os trechos codificados em SQL são subtrechos dos elementos escritos em PowerScript e dos objetos DW.

---

[3] Sistema gerenciador de banco de dados da Microsoft
[4] Sistema gerenciador de banco de dados da Oracle
[5] Linguagem do Microsoft SQL Server
[6] Linguagem do banco de dados da Oracle

3.3.3.3 *Analisador semântico*

Linguagens de alto nível são linguagens possuem alto nível de abstração (Cifuentes et al., 1998). No caso desta pesquisa se fez necessário a compilação entre linguagens de alto nível o que gera dificuldades quanto às diferenças semânticas para expressões aparentemente similares.

O PowerScript possui particularidades que não estão presentes no C#. Estas diferenças tornam o processo de migração mais complexos pois somente as expressões identificadas no código pelo analisador sintático não são suficientes para a correta tradução para elementos similares no C#.

O PowerBuilder possui escopo global para funções e variáveis, faz coerção automática de tipos de maior precisão para de precisão menor o que não ocorre no C#. Devido a estes tipos de diferenças é necessário não só conhecer os comandos no nível de sua sintaxe, mas também no nível de sua semântica, ou intensão.

A figura 13 mostra uma variável do tipo *double* recebendo o valor 1,95 (um virgula noventa se cinco) e em seguida sendo atribuída a uma variável do tipo *integer*. No PowerBuilder é feita a coerção automática entre estes dois tipos, resultando em perda de valores pois a variável do tipo *integer* receberá apenas a parte inteira do valor, ou seja, 1 (um).

```
integer valor
double codigo

codigo = 1.95
valor = codigo
```

Figura 13 - Coerção automática do PowerBuilder

No PowerBuilder funções globais definidas pelo usuário e funções globais de sistema possuem o mesmo escopo (global). Funções, tanto de sistema quanto definidas pelo usuário,

não possuem escopo global no C#. Desta maneira estas funções (tanto global quanto de sistema) devem ser diferenciadas no C# a fim de definir um escopo correto para as mesmas.

O gerador de código alvo no momento da geração do código deve ter condições de discernir se a chamada de uma função está se referindo a uma função de sistema, a uma função global definida pelo usuário, ou até mesmo a um método do objeto corrente. O gerador ainda deverá ter condição de intervir nos momentos corretos para explicitar uma coerção de tipos que no PowerBuilder é permitido que seja implícita, porém no C# não.

Para que o gerador tenha condição de fazer ajustes de percurso, com o fim de manter a mesma semântica entre os dois programas, fonte e alvo, é necessária a criação de um analisador semântico.

O Analisador semântico irá percorrer a árvore AST gerada pelo analisador léxico-sintático do PowerScript mapeando semanticamente cada uma das estruturas do código. Todo trecho de código que remete a informação (variáveis, expressões, chamadas de função) terão o tipo de dado mapeado em um modelo semântico do código (Parr, 2009), de forma que toda vez que o analisador precisar saber o tipo resultante de um trecho de código ele tenha essa informação.

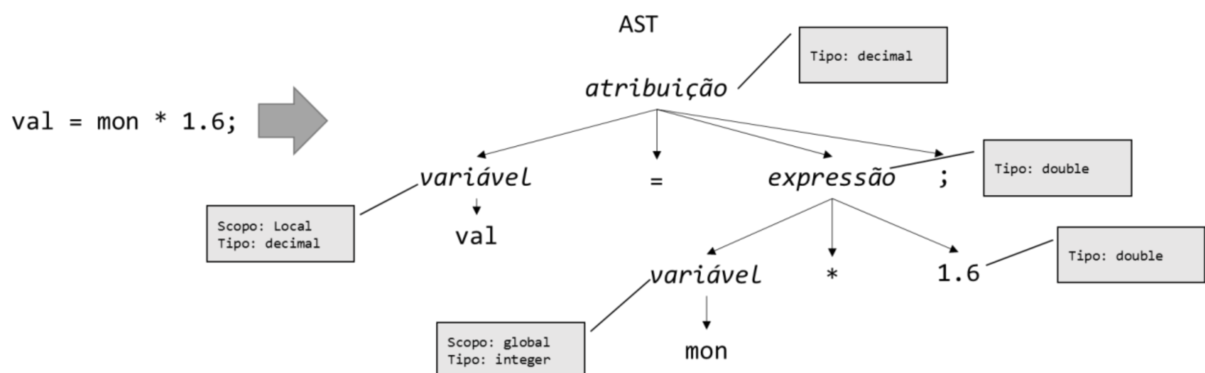

Figura 14 - Marcação semântica de uma AST

Com cada nó da arvore, que represente informação, marcado com informações semânticas o gerador pode tomar decisões a fim de manter a mesma semântica. A figura 14

mostra uma atribuição de uma expressão a uma variável. Ao analisar o tipo da variável e o tipo resultante da expressão que está sendo atribuída, pois os tipos são diferentes, o gerador faz as conversões ou coerções explicitas na linguagem alvo a fim de manter a semântica e não quebrar as regras da linguagem.

Ainda na Figura 14 é possível observar informações referentes ao escopo das variáveis envolvidas, possibilitando ao gerador identificar variáveis de escopos incompatíveis e tomar ações a respeito. O código descrito na Figura 14 poderia ser traduzido para C# conforme descrito na Figura 15. Forçar uma conversão explícita entre os tipos *double* e *decimal* e definir de onde virão as variáveis globais são vantagens que informações semânticas podem fornecer ao gerador de código alvo.

```
val = Convert.ToDecimal(Globals.mon * 1.6);
```

Figura 15 - Exemplo de código convertido

Fonte: creiadp pelo autor

3.3.3.4 Biblioteca PowerBuilder.Net

PowerBuilder e C# são ambas linguagens de alto nível. Ambas possuem expressões idiomáticas com alto nível de abstração, os controles DataWindow [7]e objetos Datastore[8] e os comandos SQL embutidos no PowerScript são exemplos no PowerBuilder. Estes não possuem correspondentes próximos na linguagem C#. Outro exemplo clássico são os arrays (estruturas homogêneas) que no C# são indexadas a partir de 0 (zero), e possuem tamanho fixo enquanto no PowerBuilder são dinâmicos e indexados a partir de 1 (um).

[7] Controles DataWindow são controles visuais que permitem interagir com objetos DataWindow® de dentro de códigos escritos em PowerScript além de gerenciar a interação dos usuários com os dados gerenciados por este controle. (Sybase, 2012a)

[8] Objetos Datastore são objetos do PowerBuilder que permitem interagir com objetos DataWindow® de dentro do código escrito em PowerScript. (Sybase, 2012b)

Inúmeras diferenças entre PowerBuilder e C# inviabilizam a tradução direta, pois muitos recursos não estão disponíveis.

As diferenças entre linguagens de alto nível não impactam somente linguagens muito distantes como PowerBuilder e C#. Mesmo a migração de Java para C# não é trivial devido às plataformas que suportam ambas as linguagens (El-Ramly et al., 2006).

Um mapeamento entre as APIs (Application Programming Interface) da linguagem fonte e da linguagem alvo pode ser feito de forma a possibilitar o gerador de código alvo a fazer as devidas traduções (Nguyen, Nguyen, & Nguyen, 2016). Esta técnica foi implementada nesta pesquisa e pode ser vista no ANEXO II.

Algumas APIs da linguagem fonte podem possuir interfaces tão diferentes da linguagem alvo que o trabalho de tradução que deveria ser feito no gerador de código alvo tornaria a construção do compilador inviável.

Este problema pode ser mitigado construindo classes na linguagem alvo que simulem a interface da linguagem fonte, possibilitando tradução direta entre estas APIs (Verhoef & Terekhov, 2000).

Portando foi necessário construir uma biblioteca de classes escrita em C# com o objetivo de aproximar alguma construção do PowerBuilder que esteja muito distante do C#. Este processo além de reduzir a complexidade do compilador melhorar em alguns casos a legibilidade do código gerado, indo de acordo com o objetivo específico 1 desta pesquisa. Este processo também é chamado de emulação de tipo e será abordado novamente na seção Síntese dos desafios.

3.4 Conversão Automatizada

Para testes do compilador e visando atender aos objetivos específicos 3, 4 e 5 foram feitas a migrações, de duas aplicações utilizando o compilador criado na etapa Construção do compilador.

A empresa onde a pesquisa foi executada possui aproximadamente 240 programas a serem migrados que juntos somam mais de 16.000.000 (dezesseis milhões) de linhas de código. Estes programas são responsáveis por gerenciar regras de negócio referente a gestão de carteira de crédito de bancos. Especificamente para esta pesquisa foram utilizados programas de um sistema específico de gestão de fundos de recebíveis. Este sistema possui 100 (cem) programas elegíveis a serem convertidos pelo compilador. São programas em PowerBuilder e sem interface gráfica.

Foi executada análise do código para verificar a diversidade de comandos de PowerBuilder existentes nos 100 programas elegíveis para migração. Como pode ser visto no quadro 1, o resultado foi um pouco surpreendente. Os programas possuem as mesmas quantidades de construções.

Quadro 1 - Resultado consolidado da análise dos fontes de programas candidatos a migração

Percentual de programas	Construções
80%	150
16%	136
4%	138

Os programas seriam selecionados para migração foram baseando-se na variedade de construções em PowerScript. Neste caso foi incluído também a quantidade de linhas como critério de desempate. A tabela 1 demostra os 10 (dez) primeiros colocados.

Tabela 1 – 10 primeiros colocados na análise do fonte

Programa	Diversidade de construções	Linhas de código
remcli	150	124.160
sfrintsac	150	98.231
sfrrelbatch	150	93.616
retcor	150	92.401
sfrremfdo	150	90.898
retcli	150	86.961
sfrconciliador	150	86.036
remcor	150	86.020
sfrretdata	150	85.214
sfrapropori	150	85.021
		Total de linhas: 928.558

Para esta análise foi utilizado o analisador sintático e semântico, construídos no passo 3.3 Construção do compilador. Eles foram executados nos códigos fonte do sistema catalogando a quantidade de cada construção do PowerScript. Os programas que, juntos, representaram a maior variedade de funcionalidades da linguagem de origem e maior quantidade de linha de código (remcli e sfrintsac) foram selecionados para migração e foram utilizados no restante da pesquisa. Este critério visou tornar o experimento mais abrangente. Ao todo foram contados 8.460.600 (oito milhões quatrocentos e sessenta mil e seiscentas) linhas de código a serem migrados neste sistema.

3.4.1 remcli

O programa "remcli" (Remessa de Cliente) valida dados lógicos enviados no arquivo de troca eletrônica entre os bancos. Dados como: campos obrigatórios (valor do título, data de vencimento), informações corretas (dígitos de CPF ou CNPJ), unicidade de títulos, valores positivos – são validados pelo programa. O programa não valida os dados diretamente no arquivo, apenas valida os dados importados previamente por outro programa. Este programada é essencial para o processo de todo o sistema e é utilizado por todos os clientes (usuários do sistema) para cessões e baixas de títulos.

A migração do programa "remcli" ocorreu da forma esperada. Foram encontrados 13 erros de compilação e 535 erros de funcionalidades não migradas. Os erros estão presentes em 590 linhas de código (alguns erros representam mais de uma linha) contra um total de 124.160 linhas totais do código original representando 99,5% de linhas de código migradas.

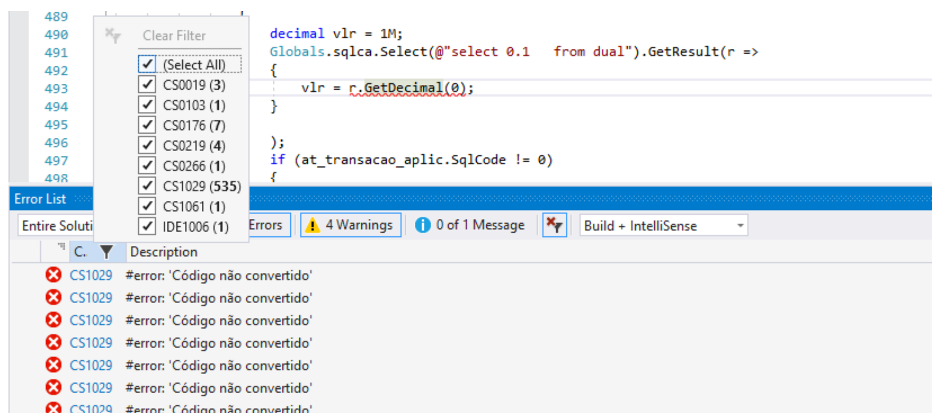

Figura 16 - Relação de erros de compilação no fonte migrado da remcli

3.4.2 sfrintsac

O programa “sfrintsac” (Integração Sistema de Carteiras), envia informações da carteira para o Sistema de Carteiras da custódia que é responsável por calcular a rentabilidade do fundo e suas respectivas cotas. São enviados 3 informações cruciais sobre a carteira, movimento (liquidações e cessões), posição (valor presente e de face da carteira) e PDD (provisão de devedores duvidosos). Esse programa é de alta criticidade para os negócios porém só é utilizado por cliente (usuários do sistema) que fazem a custódia do mesmo do fundo outros clientes não fazem a integração entre sistemas e utilizam relatórios para obter as informações.

A migração do programa “sfrintsac” também ocorreu da forma esperada. Foram encontrados muito mais erros que na migração da “remcli”, porém a maior parte deles é por uma característica do programa. Foram encontrados 436 erros de compilação e 318 erros de funcionalidades não migradas. A maior parte dos erros de compilação é devido a uma característica do programa que abre uma janela (interface visual) e a manipula diversas vezes no código. Como descrito na seção Construção do compilador interfaces visuais não serão migradas, portanto, a janela referenciada não existe no código fonte migrado, este erro representa 64% das linhas de código não migradas por erro de compilação (e não por funcionalidade não migrada).

Os erros estão presentes em 788 linhas de código contra um total de 98.231 linhas totais do código original representando 99,2% de linhas de código migradas.

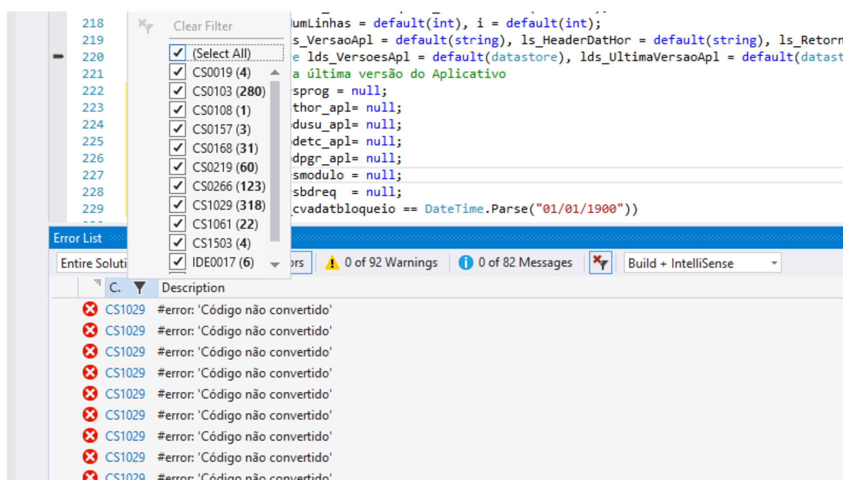

Figura 17- Resultado da compilação do fonte migrado da rotina sfrintsac

3.5 Workshop

Após a conversão dos dois programas na etapa Conversão Automatizada foi realizado um workshop com programadores da empresa onde a pesquisa foi executada. A empresa possui cerca de 160 programadores. Para que estes possam participar da pesquisa devem possuir conhecimento nas linguagens de origem e alvo, PowerBuilder e C# respectivamente. Foram convocados 24 programadores para participarem do workshop. Para exemplificar a amostra, foi feita uma classificação quanto a senioridade destes.

Tabela 2 - Nível de Senioridade dos programadores participantes do Workshop

Nível de Senioridade	Quantidade
Sênior	8
Pleno	9
Junior	7
24 programadores	

Os participantes avaliaram a qualidade do código gerado através de um questionário e estimaram o esforço necessários de adaptações dos códigos não migrados pelo compilador. Este workshop foi executado para atender aos objetivos específicos 3 e 4.

3.5.1 Qualidade do código fonte gerado

Normalmente compiladores geram códigos que não serão mantidos pois as evoluções e manutenções continuam sendo feitas no código da linguagem fonte (Chisolm & Lisonbee, 1999). Neste projeto, o código migrado passa a ser mantido e o fonte original será descartado. Portanto faz-se necessário avaliar a qualidade do código gerado (conforme objetivo específico 3). O código gerado deve:

- Estar legível
- Permitir manutenções corretivas
- Permitir manutenções evolutivas
- Possuir o mesmo desempenho do código fonte original
- Possuir a mesma semântica do código original

Os programadores responderam um questionário para validação do código gerado. Foram apresentados pares de código original/migrado e o programadores avaliaram a qualidade do código respondendo cinco perguntas sobre cada par de código utilizando a escala Likert (Likert, 1932). Foram exibidos dez trechos de códigos diferentes e os programadores responderam cinco itens a seguir sobre cada par de código:

1. **“Este código após a migração está legível”**
2. **“Este código após a migração pode ser mantido”**
3. **“Este código após a migração pode ser evoluído”**
4. **“Este código após migração está escrito de forma que não tenha perda de desempenho em relação ao original”**
5. **“Este código após a migração manteve a semântica original”**

As respostas possíveis estavam na escala a seguir:

1. Não concordo totalmente

2. Não concordo parcialmente
3. Indiferente
4. Concordo parcialmente
5. Concordo totalmente

Foram selecionados 10 trechos migrados para fazer parte do questionário de qualidade. Os trechos foram selecionados levando em consideração as transformações existentes em cada um. Os trechos selecionados podem ser vistos no ANEXO I deste trabalho.

O resultado da aplicação do questionário demonstrou boa aceitação do código gerado por parte dos programadores. O resultado detalhado será apresentados na seção 4 - Resultados.

3.5.2 Estimativa de intervenção manual

É esperado que o compilador não converta 100% do código do programa fonte para a linguagem alvo. Intervenções manuais no código do programa alvo serão necessárias para que o programa alvo possa ser executado. Assim, afim de atender ao objetivo específico 4 desta pesquisa, programadores de diferentes níveis de senioridade definiram em conjunto uma estimativa de esforço (homem/hora) necessários para finalizar os programas convertidos. Foram selecionados nove profissionais da amostra anterior para participarem desta etapa, três de cada nível de senioridade (sênior, pleno e júnior).

Foi feita uma pré-análise no código fonte gerado com o objetivo de buscar pelas construções da linguagem origem não convertidas e que consequentemente necessitarão de intervenção manual. Estas construções foram parcialmente exibidas na seção Conversão Automatizada. Os participantes foram reunidos em uma sala onde foram exibidos trechos de código para que pudessem estimar o esforço necessário para a adequação da funcionalidade.

O processo seguiu a metodologia de planning poker (Grenning, 2002) que permite fazer estimativas em grupo de forma empírica. (Haugen, n.d.; Molokken-Ostvold & Haugen, 2007)

demostraram estudos do uso do planning poker para estimativas de esforço homem/hora para desenvolvimento. O objetivo era que a cada trecho observado pelo grupo se chegasse um consenso sobre o esforço homem hora para o item específico em análise.

Os resultados detalhados serão apresentados na seção 4 - Resultados.

3.6 Síntese dos desafios

(Verhoef & Terekhov, 2000) falam em seu trabalho sobre o processo de conversão automatizada de código fonte. Eles descrevem a respeito de vários fracassos neste tipo de projeto. O trabalho expõe diversos problemas que podem ocorrer nesta automatização, a maior parte delas relacionadas ao sistema de tipos das linguagens, que podem gerar diferenças semânticas entre as aplicações. Deixam claro no fim do trabalho que o processo não é trivial e que a organização tem que estar alinhada quanto às expectativas deste tipo de projeto.

Devido a estas dificuldades foram listados os principais desafios encontrados durante a execução do processo de automatização da conversão destes códigos fonte. São considerados desafios quaisquer elementos que podem reduzir a quantidade de código migrada, reduzir qualidade do código gerado ou aumentar o custo do processo. Estes desafios podem inviabilizar o processo tecnicamente ou financeiramente e a síntese dos desafios encontrados neste processo podem contribuir para trabalhos futuros.

Os desafios identificados foram separados pela fase do processo ao qual se aplicam: análise léxico-sintática, análise semântica e transformação.

3.6.1 Classificações dos Desafios

Os desafios da fase de transformação foram subclassificados de acordo com a complexidade e natureza de sua solução. A seguir estão descritas as classificações e quais critérios determinam os desafios que farão parte de cada uma delas. As classificações foram

retiradas do trabalho de (El-Ramly et al., 2006) e (Verhoef & Terekhov, 2000) e serão utilizadas ao descrever os desafios da fase de transformação.

3.6.1.1 Transformação Direta

Um desafio é classificado como transformação direta quando existe uma relação um para um do PowerBuilder para C#, onde apenas um processo de substituição simples de um é suficiente para fazer a transformação. As soluções dadas por transformação direta não necessitam de informações semânticas dependendo apenas da análise léxico-sintática para serem implementadas.

3.6.1.2 Transformação Indireta

Um desafio é classificado como transformação indireta quando alguma inteligência é necessária para mapear uma construção do PowerBuilder para o C#. O processo de transformação direta apesar de envolver uma certa inteligência não exige algoritmos de transformação complexos além do código final não estar muito diferente do código original.

3.6.1.3 Transformação Desafiadora

Um desafio é classificado como transformação desafiadora quando um processo complexo de transformação está envolvido, ou quando o resultado da transformação difere muito do código original. As transformações desafiadoras são as que mais trarão contribuição para projetos futuros.

3.6.1.4 Emulação de Tipo

Um desafio é classificado como emulação de tipo quando uma transformação desafiadora se torna tão complexo que poderia inviabilizar tecnicamente ou financeiramente o projeto de automatização da conversão de linguagens. A emulação de tipo consiste em criar uma estrutura na plataforma da linguagem alvo, deforma a simplificar o processo de conversão.

Emulação de tipo pode tornar transformações desafiadoras em transformações indiretas ou até mesmo em transformações diretas. Este processo é o último recurso para solução de um desafio. Nesta pesquisa foram dadas preferências para transformações sendo a emulação de tipo preterida como solução dos desafios encontrados.

3.6.2 Desafios de Análise Léxico-sintática

Os desafios desta parte estão restritos à análise do código, independentemente de suas notações semânticas ou necessidades de transformação. Os desafios encontrados nesta fase são relacionados a compreensão sintática do código. Alguns destes desafios podem se aplicar a outras linguagens. Portanto os desafios a seguir estão relacionados à compreensão, do programa analisador (parte do compilador) das linguagens fontes envolvidas, no caso deste trabalho: PowerScript, SQL e DataWindow.

3.6.2.1 Case Insensitive

Problema: Tanto o PowerScript quanto o SQL são linguagens cujo o *case* dos comandos não fazem diferença para o processamento léxico. Ao procurar por um comando “if” esse pode estar escrito com qualquer combinação de caixa alta ou baixa de seus caracteres. Ou seja, if, IF, iF ou If são aceitos como comando “if” no PowerScript, assim como no SQL os comandos podem ser escritos de qualquer forma. O ANTLR4 por padrão gera reconhecedores *case sensitive,* ou seja, onde o *case* dos caracteres diferencia uma construção léxica de outra.

Solução: Para o problema supracitado foram dadas duas soluções distintas. No PowerScript foi criado um interceptador no analisador léxico. Foi criado uma especialização da classe *AntlrInputStream*, que durante o reconhecimento dos tokens, transforma todos os caracteres para minúsculos. Esta modificação não altera o conteúdo do que será processado ou analisado, apenas no momento do reconhecimento dos tokens, os caracteres são reconhecidos sempre em minúsculo. A segunda forma foi utilizada na gramática de SQL foram definidos

fragmentos léxicos para cada letra do alfabeto. Como pode ser visto na figura 18, cada fragmento é utilizado para definir a os caracteres que aquele fragmento representa no caso do fragmento “A” (que representa a letra A) ele representa tanto a letra “A” maiúscula quanto a letra “a” minúscula. Assim as *keywords*, palavras reservadas, que são reconhecidas pelo analisador léxico podem ser montadas utilizando estes fragmentos, conforme demonstrado na figura 19. O comando “Select” do SQL é definido como uma construção léxica utilizando os fragmento S, E, L, C e T, permitindo reconhecer qualquer uma das variações possíveis da palavra-chave do SQL.

```
fragment A: [aA];
fragment B: [bB];
fragment C: [cC];
fragment D: [dD];
fragment E: [eE];
fragment F: [fF];
fragment G: [gG];
fragment H: [hH];
fragment I: [iI];
fragment J: [jJ];
fragment K: [kK];
fragment L: [lL];
fragment M: [mM];
fragment N: [nN];
fragment O: [oO];
fragment P: [pP];
fragment Q: [qQ];
fragment R: [rR];
fragment S: [sS];
fragment T: [tT];
fragment U: [uU];
fragment V: [vV];
fragment W: [wW];
fragment X: [xX];
fragment Y: [yY];
fragment Z: [zZ];
```

Figura 18 - Fragmentos lexicos para definir keywords como case insensitive

```
SELECT:      S E L E C T;
DISTINCT:    D I S T I N C T;
ALL:         A L L;
```

Figura 19 - Construção léxica de algumas Keywords do SQL

3.6.2.2 *Keywords and Identifiers Ambiguity*

Problema: Algumas palavras chaves (keywords) podem ser utilizadas como nome de identificadores. Neste caso o *lexer* (analisador léxico) não consegue distinguir se o token é de uma comando específico, ou se ele é um identificador, apenas escrevendo expressões regulares.

É necessário a inclusão de contexto neste ponto do programa e o ANTLR4 assim como a maioria de linguagens para geração de *parser*, não suportam nativamente linguagens não livres de contexto. Esse problema é muito incomum em linguagens de programação (Hopcroft et al., 2001).

Solução: A solução para este item foi adicionar um código especial ao programa de reconhecimento do PowerScript. Este código faz análise dos *tokens* envolvidos anteriormente ou posteriormente ao *token* que está sendo analisado e muda o seu tipo para identificador. Na figura 20 a regra léxica para a palavra chave “readonly”, que determina que um parâmetro está sendo passado como somente leitura, ou atributo está sendo declarado apenas para leitura, porém, o *token* “readonly” pode ser utilizado como nome de atributo ou nome de métodos em algumas ocasiões, nesse caso ele deve ser considerado não mais como a palavra-chave “readonly” mas sim um “identificador”.

```
READ_ONLY   :   {var texto = InputStream.ToString().Substring(0, this.InputStream.Index);
            var inicioLinha= texto.LastIndexOf(".");
            texto = inicioLinha >= 0 ? texto.Substring(inicioLinha): "";
            if (texto.Trim() == ".")
               dotExpression = true;
         }
             'readonly'
         {
             if(dotExpression)
                 Type=ID;

             dotExpression = false;
         }
         ;
```

Figura 20- Regra léxica com intervenção de código para resolução de ambiguidades

3.6.2.3 Line Continuing

Problema: No PowerScript a forma padrão de indicar fim de comando é a quebra de linha. Este indicador em linguagens como C#, C, C++ e Java é o caractere “;” (ponto e vírgula). Porém é possível quebrar um comando em mais de uma linha de código. Para isso utiliza-se o caracter “&” (e comercial) para identificar que apesar de haver uma quebra de linha logo após

o “&” ela não deve ser considerada fim de comando e deve ser desprezada. Este problema não impacta as regras léxicas porém as regras sintáticas são todas impactadas, pois uma quebra linha pode ser introduzida entre qualquer *token* presente na regra sintática sendo previsto na regra ou não.

Solução: Foi criada uma regra léxica enviando a sequência “&” seguido por quebra de linha para o canal “*Hidden*” assim não farão parte das regras sintáticas. A figura 21 mostra como foi implementada a regra léxica. WS_1 representa 0 (zero) ou mais espaços em branco, enquanto NL representa uma ou mais quebras de linha.

```
LC : E_COMM WS_1 NL WS_1
{
    Channel=Hidden;
}
;
```

Figura 21 - Definindo Line Continuing para canal oculto (Hidden)

3.6.2.4 Line Comments

Problema: Comentários de linha terminam com quebra de linha. Quando comentários de linha são inseridos na mesma linha que comandos (após os comandos) a quebra de linha passa a ter duas funções: fim de comentário, e fim de comando.

Solução: Foi criado um código especial para o reconhecedor léxico, de forma a emitir um novo delimitador (simulando como se houvessem dois delimitadores seguidos) para que um possa fazer parte da regra léxica do comentário de linha e o outro possa fazer parte da regra sintática do comando. Para isso foi necessário intervir com código a ser executado pelo analisador léxico codificando a emissão do delimitador de comandos. A figura 22 demostra a codificação necessária para resolver o problema. Primeiro verifica-se se antes do comentário de linha existe algum comando e então cria-se um novo *token* e este é inserido no fluxo de *tokens* para ser consumido pela regra sintática do comando que antecede o comentário.

```
LINE_COMMENT
    :   LINE_COMMENT_START ~('\n'|'\r')*? NL NOTHING
    {

        Channel=Hidden;

        //Para comentários que começam após um trecho de código deve-se emitir o delimitador
        //para marcar o fim do código anterior ao comentário
        if (Line >= 2)
        {
            //var textoLinha = this.InputStream.ToString().Split('\n')[this.Line - 2];
            var linhas = this.InputStream.ToString().Split('\n');
            var textoLinha = string.Empty;
            for (int i = this._tokenStartLine - 1; i < this.Line - 1; i++)
                textoLinha += linhas[i];

            if (textoLinha != "\r" && textoLinha.Substring(0, textoLinha.IndexOf("//")).Trim().Length > 0)
            {
                var newToken = new CommonTokenDecorator(this.TokenFactory.Create(DELIMIT, ""));
                newToken.Channel = PBLexer.DefaultTokenChannel;
                newToken.Line = Line;
                newToken.Column = Column;
                this.InsertToken(newToken);
                //System.Console.WriteLine("Emitndo DELIMIT");
            }
        }
        //this.Skip();
        //this.Emit(this.TokenFactory.Create(DELIMIT, "\r\r"));
        //System.Console.WriteLine("Comentario de Linha (Match 2): {0}", Text);
    }
    ;
```

Figura 22 - Definição lexica para comentários de linha resolvendo ambuiguidade do delimitador de comandos

3.6.3 *Desafios de Transformação*

A transformação é o processo de converter um código fonte em outro de mesmo nível (El-Ramly et al., 2006). Neste caso estamos transformando de uma linguagem alto nível para outra linguagem de alto nível.

Os desafios encontrados na fase de transformação serão apresentados a seguir relacionando o problema e sua solução e a classificação da solução.

3.6.3.1 *Choose case*

Problema: O Comando "Choose case" do PowerBuilder generalizadamente possui a mesma função que o comando switch do C#. Porém o PowerBuilder permite uma série de exceções na execução do comando, o que não permite uma transformação direta.

Solução: Os comandos "choose case" são substituídos por comandos "if" aninhados mantendo a semântica original do código.

Classificação: Transformação desafiadora.

Original	Convertido

<pre>choose case ll_ret
    case 0
        msg = "OK"
    case 1 to 13
        msg = "Erro de rede"
    case else
        msg = "Erro desconhecido"
end choose</pre>
	<pre>if (ll_ret == 0)
	    msg = "OK";
	else if (ll_ret >= 1 && ll_ret <= 13)
	    msg = "Erro de rede";
	else
	    msg = "Erro desconhecido";</pre>

Figura 23 - Exemplo de transformação do Choose Case

3.6.3.2 Embedded SQL

Problema: Comandos SQL podem ser inseridos ao longo do código PowerScript no PowerBuilder. Estes comandos permitem acessos diretos ao banco de dados sem a utilização de componentes ou objetos específicos.

Solução: Os comandos de DML (Data Maintaining Language) e DRL (Data Retrieve Language) são executados através de um tipo emulado. Mesmo utilizando o tipo emulado este desafio demanda uma transformação desafiadora pois necessita que o comando SQL seja analisado de forma que possa interagir de forma coerente com o código C#. Pelo fato de envolver variáveis código, onde são lidos os argumentos e retornados os dados, envolvidas nos comandos SQL os comandos precisam passar por esta análise e serem transformados de forma a repassar cada uma de suas variáveis para o tipo emulado, que as utiliza tanto para entrada como para saída.

Classificação: Emulação de Tipo e Transformação Desafiadora

Original

```
select count(1)
into :ldb_count
from tb_titlo, tb_ctlcart
where tb_titlo.ctlcart_cod = tb_ctlcart.ctlcart_cod
and tb_titlo.emp_cod = :gl_emp
and tb_titlo.titlo_datmov = :adt_datatu
and tb_titlo.titlo_dat_incl = :adt_datatu
and ( tb_titlo.titlo_sit_contabil is null OR tb_titlo.titlo_sit_contabil <> 'S' )
and tb_ctlcart.ctlcart_tip = 'C'
and tb_titlo.ctlcart_cod <> 81;
```

Convertido

```
Globals.sqlca.Select(@" select count(1)
                        from tb_titlo, tb_ctlcart
                        where tb_titlo.ctlcart_cod = tb_ctlcart.ctlcart_cod
                        and tb_titlo.emp_cod = :v1
                        and tb_titlo.titlo_datmov = :v2
                        and tb_titlo.titlo_dat_incl = :v3
                        and ( tb_titlo.titlo_sit_contabil is null OR tb_titlo.titlo_sit_contabil <> 'S' )
                        and tb_ctlcart.ctlcart_tip = 'C'", gl_emp, adt_datatu, adt_datatu)
.GetResult(r =>
{
    ldb_count = r.GetDecimal(0);
}
```

Figura 24 - Exemplo de transformação do SQL Embbeded

3.6.3.3 1 Based Arrays

Problema: Arrays (vetores) no PowerBuilder inicia-se por padrão do índice 1 enquanto no C# são iniciados a partir do índice 0 (zero).

Solução: Os acessos a índices de vetores ao serem transformados para C# são subtraídos em 1 (um). Expressões recebem uma subtração de 1 (um) e valores literais são transformados em tempo de compilação para valor subtraído de 1 (um).

Classificação: Transformação indireta

Original

```
//Busca as dependências
objAPI.f_getdepend(lEmpCod[i], lDepCod, sDepDes)

for j=1 to upperbound(lDepCod)
    lLinha++ //incrementa o indice do vetor
    as_emp_cod[lLinha] = string(lEmpCod[i])
    as_dep_cod[lLinha] = string(lDepCod[j])
next
```

Convertido

```
//Busca as dependências
objAPI.f_getdepend(lEmpCod[i - 1], ref lDepCod, ref sDepDes);
for (j = 1; j <= lDepCod.Length; j++)
{
    lLinha++; //incrementa o indice do vetor
    as_emp_cod[lLinha - 1] = lEmpCod[i - 1].ToString();
    as_dep_cod[lLinha - 1] = lDepCod[j - 1].ToString();
}
```

Figura 25- Exemplo de transformação do 1 Based Array

3.6.3.4 Dynamically Allocated Arrays

Problema: Arrays (vetores) no PowerBuilder são dinamicamente alocados. Por padrão arrays não possuem limite superior. No C# vetores possuem limite superior definido no momento da alocação de memória do vetor.

Solução: Criação de um tipo emulado que simula um vetor do C#, porém com alocação dinâmica. Este problema foi identificado por (Kontogiannis et al., 2010) cuja solução proposta foi a mesma deste trabalho, emulação de tipo.

Classificação: Emulação de Tipo.

3.6.3.5 Function Call

Problema: PowerBuilder possui funções globais, métodos que não estão vinculados a classe alguma. No C# não existem funções, apenas métodos. Métodos sempre devem estar vinculados a uma classe.

Solução: Ao gerar o código as funções são agrupadas em classes estáticas. De forma a possibilitar que o código C# consiga executá-las. O trabalho desta solução está em identificar semanticamente através da tabela de símbolos a qual tipo de função global ela se refere, função de sistema ou do usuário. Um detalhamento maior desta solução pode ser visto na solução dos desafios: Global System Functions e Global User Functions.

Classificação: Transformação Indireta/Emulação de tipo

3.6.3.6 Initial Value of Variables

Problema: No PowerBuilder variáveis de tipo primitivo são inicializadas automaticamente com seus valores padrões no momento da declaração. No C# variáveis não podem ser utilizadas sem terem sido inicializadas.

Solução: Na declaração de variáveis, todas elas são iniciadas com seu valor padrão, utilizando o comando default do C#.

Classificação: Transformação direta.

Original	`string ls_sql` `Integer li_sgbd, li_retorno`
Convertido	`string ls_sql = default (string);` `short li_sgbd = default (short), li_retorno = default (short);`

Figura 26 - Exemplo de transformação do Initial Value of Variables

3.6.3.7 Primitive Type Date

Problema: PowerBuilder possui um tipo primitivo chamado "date" que representa apenas data sem a hora. O tipo mais próximo no C# é o DateTime que representa Data e Hora conjugados.

Solução: Foi criado um tipo chamado Date que encapsula o tipo de dado DateTime do C#. Este tipo garante a manipulação apenas da parte data do tipo de dado do DateTime.

Classificação: Emulação de Tipo.

3.6.3.8 Primitive Type Time

Problema: PowerBuilder possui um tipo primitivo chamado "time" que representa apenas hora sem a data. O tipo mais próximo no C# é o DateTime que representa Data e Hora conjugados.

Solução: Foi criado um tipo chamado Time que encapsula o tipo de dado DateTime do C#. Este tipo garante a manipulação apenas da parte hora do tipo de dado do DateTime.

Classificação: Emulação de Tipo.

3.6.3.9 File Manipulation

Problema: O PowerBuilder possui funções globais de sistema para manipulação de arquivos. O C# possui classes para o mesmo propósito. A interface das plataformas é muito incompatível.

Solução: As funções de arquivos do PowerBuilder foram emuladas em conjunto com as demais funções de sistema sem correspondência no .Net Framework.

Classificação: Emulação de Tipo/Transformação Direta.

Original	`ll_ret = FileWriteEx(il_arquivo, as_texto)`
Convertido	`ll_ret = SystemFunctions.FileWriteEx(il_arquivo, as_texto);`

Figura 27 - Exemplo de transformação do File Manipulation

3.6.3.10 Global System Functions

Problema: O PowerBuilder possui funções globais de sistema estas chamadas devem ser diferenciadas das chamadas de funções globais do usuário.

Solução: Foi criada classe estática chamada SystemFunctions onde todas as funções globais do PowerBuilder (sem correspondência com o .Net Framework) são emuladas. Para maior legibilidade do código as funções globais de sistema foram inseridas como importação estática no C#, como demonstrado na figura 28, o que permite que elas sejam chamadas da mesma forma que no PowerBuilder sem especificar no código a classe SystemFunctions. Assim, nenhuma transformação é necessária ao acessar funções de sistema emuladas.

Classificação: Emulação de Tipo

```
using static attps.Biblioteca.PowerBuilder.Net.SystemFunctions;
```

Figura 28 – Inclusão de classe SystemFunction como importação estática

3.6.3.11 Global User Functions

Problema: O PowerBuilder possui funções globais de sistema estas chamadas devem ser diferenciadas das chamadas de funções globais do usuário.

Solução: Todas as funções globais de usuário são agrupadas em uma classe gerada durante o processo de conversão chamada GlobalFunctions. Para identificar se uma função é de usuário ou de sistema, todas as funções nativas do PowerBuilder foram catalogadas e inseridas em escopo global do analisador semântico de forma que ao encontrar estas funções o analisador às relacionava ao símbolo de funções nativas do PowerBuilder. Durante o processo de conversão, se uma função é encontrada no escopo global, esta é acessada através da classe estática SystemFunctions caso seja uma função de sistema, caso contrário, a função é acessada através da classe GlobalFunctions.

Classificação: Transformação Indireta

Original	`if sqlca.sqlcode = -1 then`
	`  f_prepara_log("Uo_atualiza", "", "f_calc_juros_vnd", 1, "Erro, Dados da Instrução!")`
	`  return "Erro"`
	`end if`

Convertido	`if (Globals.sqlca.SqlCode == -1)`
	`{`
	`    GlobalFunctions.f_prepara_log("Uo_atualiza", "", "f_calc_juros_vnd", 1, "Erro, Dados da Instrução!");`
	`    return "Erro";`
	`}`

Figura 29 - Exemplo de transformação do Global User Functions

3.6.3.12 Datastore Class

Problema: A classe datastore é uma das mais utilizadas em programas powerbuilder (não visuais) para manipulação de dados. Não existe uma estrutura de interface similar no C#.

Solução: Foi criada classe chamada DataStore que emula o funcionamento da Datastore do PowerBuilder. A interface replicada consiste nos elementos mais utilizados.

A emulação da datastore é um dos processos mais complexos deste trabalho. O objeto é muito rico em funcionalidades e a replicação de todos os seus comportamentos em C# é trabalhoso e arriscado. A emulação deste tipo foi responsável por 15% de todo o esforço necessário para a execução deste trabalho.

Classificação: Emulação de Tipo.

3.6.3.13 Datawindow Objects

Problema: Os objetos Datawindow são instruções para classes datastore de como recuperar e atualizar informações e no banco de dados além de definir a estrutura interna dos objetos da classe datastore. Não existe recurso similar no C#.

Solução: Foi criada um tipo (DataObject) para emular os objetos DataWindow Objects que são analisados pelo *parser* as informações extraídas dão origem a uma classe herdada de DataObject (classe emulada) contendo todas as informações para o funcionamento correto da DataStore.

Classificação: Emulação de Tipo.

3.6.3.14 Transaction Class

Problema: A classe Transaction representa a conexão e, se existir, uma transação com o banco de dados. Os comandos SQL Embedded são executados através de um Transaction. Não existe objeto de estrutura similar no PowerBuilder.

Solução: Foi criada classe Transaction que permite utilizar uma conexão padrão .Net Framework (que implementa interface IDbConnection) para acessar o banco de dados. A classe foi munida de dois métodos: Execute e Select, para executar comandos de DML e DRL respectivamente. Estes métodos serão responsáveis por emular a execução de comandos SQL

Embedded, exemplos podem ser visualizado no desafio “Embedded SQL”. Além dos métodos para execução de conexão foram implementados métodos para gestão de transações.

Classificação: Tipo Emulado/Transformação Indireta.

Original	`if ll_ret = -1 then`
	`rollback;`
	`return -1`
	`end if`

Convertido	`if(ll_ret == -1)`
	`{`
	`Globals.sqlca.Rollback();`
	`return -1;`
	`}`

Figura 30 - Exemplo de transformação do Transaction Class

3.6.3.15 Case Insensitive

Problema: O PowerScript é *case insensitive* para os comandos e para os identificadores (nomes de variáveis, métodos e classes), portanto os elementos podem ser declarados com letras caixa alta e utilizados com letras em caixa baixa. O C# é *case sensitive*, neste caso todo acesso aos identificadores tem de ser feito conforme (*case*) definido em sua declaração.

Solução: Durante o processo de conversão, ao interceptar um identificador (nome de método, variável, classes e atributos) é feito acesso à tabela de símbolos de forma a obter a formatação do identificador em sua definição. Esta formatação é utilizada para referenciar o identificador no código convertido.

Classificação: Transformação Indireta.

3.6.3.16 Automatic Number Casting

Problema: O PowerBuilder por padrão não faz checagem de precisão entre tipos em atribuição de tipos compatíveis como "int" e "long" além de não gerar erro de "*aritmetic overflow*". O C# gera erro nas atribuições entre tipos de maior precisão em menor precisão.

Solução: Durante o processo de conversão, ao interceptar uma operação de atribuição, ou passagem de parâmetros é feita análise da tabela de símbolos quanto ao tipo de dados envolvidos na operação. Caso sema operações entre números é feita consulta à tabela de compatibilidade entre tipos (criada para resolver este problema). Através do acesso a esta tabela é avaliada a necessidade de inserir uma operação de casting, fazendo com que o C# permita a execução da operação entre estes números.

Classificação: Transformação Indireta.

Original	`li_index = ll_index + 1`
Convertido	`li_index = (short)(ll_index + 1);`

Figura 31 - Exemplo de transformação do Automatic Number Casting

3.6.3.17 Char and String Literals

Problema: No PowerBuilder valores literais de *String* e *Char* podem ser feitos tanto com aspas simples quanto com aspas duplas. No C# *string* são representadas com aspas duplas e *char* com aspas simples.

Solução: Ao identificar literais de string é feita consulta à tabela de símbolos de forma a identificar o tipo do destino do literal. Caso o destino seja um char e o literal seja uma *string* o mesmo é transformado para apenas um caractere *char*. Caso seja o contrário, o literal de *char* é transformado em literal *string*. Caso a operação seja de uma expressão *string* para um destino *char* é utilizado operador de indexação do C# ([] colchetes) obtendo apenas o primeiro caractere da *string* e a retornando para a operação.

Classificação: Transformação Indireta.

Original	`lc_tipo = ls_tipo`
Convertido	`lc_tipo = ls_tipo[0];`

Figura 32 - Exemplo de transformação do Char and String Literals

3.6.3.18 Automatic Casting

Problema: No PowerBuilder não existe operador para *cast* entre tipos. Em atribuição de objetos da mesma família (ancestrais e descendentes) não é necessário especificar um *cast* ao atribuir um valor de uma variável de um tipo mais genérico à uma variável de um tipo mais específico. No C# esta operação só pode ser feita especificando operador de casting.

Solução: Ao identificar operações de atribuição ou passagem de parâmetros é feita análise dos tipos envolvidos, caso ambos sejam da mesma família (relacionados diretamente ou indiretamente via herança) é feita análise da necessidade de casting. Caso o tipo mais genérico esteja sendo atribuído à um tipo mais específico é utilizado o operador de casting do C# para permitir a execução da operação.

Classificação: Transformação Indireta.

3.6.3.19 Default Access Modifiers

Problema: No PowerBuilder atributos são públicos por padrão. No C# ao não especificar um modificador de acesso para um atributo o mesmo é considerado privado.

Solução: Ao fazer a conversão da estrutura de classes o modificador de acesso *public* é utilizado em casos onde não foi definido o modificador de acesso no código PowerBuilder.

Classificação: Transformação Direta.

3.6.3.20 Native Functions

Problema: O PowerBuilder possui uma muitas funções globais nativas provenientes de sua plataforma. O C# possui a sua própria gama de funções nativas disponibilizadas em diversas classes do .Net Framework. As interfaces são muito diferentes e existem funções não correspondidas pelo C#.

Solução: Foi feita análise das funções globais do PowerBuilder e quais tinha transformações diretas para C#. Todas estas estão sendo utilizadas diretamente do C#, as demais precisarem ser emuladas. No ANEXO II deste trabalho está descrito o mapeamento entre as funções. A primeira coluna (Função PB) descreve o nome da função nativa do PowerBuilder, a segunda (Mapeamento) descreve se/como será feita a transformação e a terceira coluna (Função C#) descreve a construção utilizada no C# no caso de o mapeamento ser feito por algum tipo de transformação.

Classificação: Emulação de Tipo

3.6.3.21 Global Variables

Problema: O PowerBuilder possui o conceito de variáveis globais. Este conceito não existe no C#.

Solução: Durante o processo de conversão é criado uma classe estática chamada “Globals”. Toda declaração de variável global é registrada como um atributo estático dessa classe. Ao identificar um acesso a variável durante o processo de conversão é consultado na tabela de símbolos qual o escopo desta variável, caso seja uma variável de escopo global o acesso a mesma é feito através da classe Globals.

Classificação: Emulação de tipo.

Original	`if gl_empresa = 1 then`
	`return "Matriz"`
	`end if`

Convertido	`if (Globals.gl_empresa == 1)` `return "Matriz";`

Figura 33 - Exemplo de transformação do Global Variables

3.6.3.22 Exponential Operator

Problema: O PowerBuilder possui operador para operações de exponenciação (^). O C# não possui esse operador.

Solução: Mapear a utilização do operador de exponenciação para função Math.Pow. A função Math.Pow executa operações de exponenciação porém possui apenas um *overload* que trabalha com o tipo *double*. Assim foi necessário a inserção de *cast* para o tipo ao qual o operador está retornado seu valor em uma expressão. Para essa transformação a operação precedente é avaliada e obtido seu tipo através da tabela de símbolos .

Classificação: Transformação indireta.

Original	`ll_bytes = ll_mega * (2 ^ 20)`
Convertido	`ll_bytes = ll_mega * (int)(Math.Pow(2, 20));`

Figura 34 - Exemplo de transformação do Exponential Operator

3.6.3.23 Integer Precision

Problema: Os tipos interios do PowerBuilder possuem precisão que os mesmos tipos no C#.

Solução: Foi feito mapeamento dos tipos e precisões e criada uma tabela simples de conversão entre inteiros. A tabela 3 demostra a relação entre os tipos.

Classificação: Transformação direta.

Tabela 3- Relação entre tipos inteiros do PowerBuilder e C#

Precisão	**Signed**	**PowerBuilder**	**C#**
8 bits	Sim	-	sbyte
8 bits	Não	Byte	byte
16 bits	Sim	Int, Integer	short
16 bits	Não	UInt, UnsignedInt, UsignedInteger	ushort
32 bits	Sim	Long	int
32 bits	Não	ULong, UnsignedLong	uint
64 bits	Sim	Longlong	long
64 bits	Não	-	ulong

3.7 Avaliação da viabilidade da automação

Pode parecer óbvio que um processo automatizado é mais eficiente que o processo manual. Autores como (Chisolm & Lisonbee, 1999; Kontogiannis et al., 2010; PIRKELBAUER, 2010) defendem automatização do processo em detrimento do processo manual. Porém o custo da construção do processo pode o inviabilizar economicamente.

Nesta etapa da pesquisa foi feita análise de viabilidade do processo automatizado, visando atender ao objetivo específico 5, tendo como insumo as informações obtidas durante a execução desta pesquisa. Indicadores coletados nas etapas anteriores foram analisados deforma a identificar o ganho ou o ônus deste processo.

O Quadro 2 demostra os indicadores que foram utilizados para a análise inicial. Os projetos da empresa são controlados por dois sistemas principais o CP (Controle da Produção) e o EMP (Sistema de gestão de projetos). O sistema de CP contém o apontamento diário de horas dos profissionais. Esta informação pode ser obtida através de relatórios do sistema. A figura 35 demostra tela de consulta de apontamento de horas e a figura 36 demostra um exemplo do relatório gerado. Todas as informações de apontamento, inclusive as da construção do compilador, foram extraídas do sistema CP.

Quadro 2- Indicadores análise processo

Código	**Indicador**	**Unidade**	**Origem**
EC	Esforço construção compilador	Homem hora (hh)	Sistema de controle da produção
CC	Custo/hora médio profissionais construção do compilador	Reais/hh	EPM – Sistema controle de projetos
EMM	Esforço para migração manual	Homem hora (hh)	Sistema de controle da produção
CMM	Custo/hora médio profissionais migração manual	Reais/hh	EPM – Sistema controle de projetos
EMA	Esforço para migração automatizada	Homem hora (hh)	Workshop etapa Estimativa de intervenção manual
CMA	Custo/hora médio profissionais migração automatizada	Reais/hh	EPM – Sistema controle de projetos
TLOC	Total de linhas de código de todos os programas	Unidade	Código fonte
TINM	Ocorrências de um item não migrado	Unidade	Código Fonte
TFPA	Total de pontos de função de todos os programas	Unidade	EPM – Sistema controle de projetos

Alguns programas já foram convertidos manualmente pelos programadores da empresa. O apontamento das horas gastas se encontra no sistema CP e foram utilizados nesta fase da pesquisa. Um programa já migrado manualmente foi selecionado para passar pelo processo automatizado para que a comparação seja feita em cima dos mesmos dados. As estimativas de horas de intervenção manual serão obtidas através de projeção dos dados obtidos na etapa Workshop.

Os campos com descrição em vermelho são de preenchimento obrigatório.
Planilha: Consultar Apropriação - De Acordo com o Project
Obrigatorio: Chamado Projeto Grupo Pessoa Produto C.Custo Subpro
Projeto: 513 Projeto FIDC - Migração ASIS
Grupo:
Produto:
Módulo:
Pessoa:
Tipo:
Cliente:
C.Custo:
Início Período: 01/10/2016
Fim Período: 31/10/2016
Enviar

Figura 35- Tela de consulta de apontamento de horas

Após a coleta dos indicadores eles foram refinados de forma a facilitar a utilização para o cálculo da viabilidade econômica. Os indicadores de CMM e EMM foram multiplicados gerando um novo indicador CTMM (Custo Total da Migração Manual) o mesmo foi feito com os indicadores CC e EC para a geração do CTC (Custo Total do Compilador) conforme pode ser visto no Quadro 3. O refinamento dos indicadores permitiu uma visão mais clara da eficiência da migração automatizada. O indicador CTC permite visualizar o custo especificamente da construção do compilador, custo que pode ser diluído em migrações posteriores a fim de se calcular o ROI[9]. O CTMM pode ser utilizado para estimar o custo de futuras migrações manuais. Ao contar a quantidade de linhas de código (LOC) dos programas podemos prever um custo provável de migração de outros programas, assim identificando quantas LOCs seriam necessárias para se pagar o investimento inicial feito no compilador (CTC).

[9] Retorno do investimento

	B	F	G	H	I	J	K	L	U
1	Profissional	Numero	Data	Hora Inicio	Hora Fim	Tempo	Tempo Decimal	Atividade	Projeto
2									
3	Alexandre Her	569453	10/10/2016 00:00:00	09:18:00	13:00:00	3:42	3,70	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
4	Alexandre Her	569453	10/10/2016 00:00:00	14:00:00	18:39:00	4:39	4,65	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
5	Alexandre Her	569453	11/10/2016 00:00:00	16:00:00	18:16:00	2:16	2,27	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
6	Alexandre Her	569453	13/10/2016 00:00:00	09:16:00	12:59:00	3:43	3,72	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
7	Alexandre Her	569453	13/10/2016 00:00:00	14:01:00	17:00:00	2:59	2,98	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
8	Alexandre Her	569453	14/10/2016 00:00:00	09:17:00	12:55:00	3:38	3,63	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
9	Alexandre Her	569453	14/10/2016 00:00:00	14:05:00	16:34:00	2:29	2,48	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
10	Alexandre Her	569453	17/10/2016 00:00:00	09:34:00	13:03:00	3:29	3,48	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
11	Alexandre Her	569453	17/10/2016 00:00:00	14:07:00	18:39:00	4:32	4,53	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
12	Alexandre Her	569453	18/10/2016 00:00:00	09:45:00	13:00:00	3:15	3,25	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
13	Alexandre Her	569453	18/10/2016 00:00:00	14:00:00	18:17:00	4:17	4,28	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
14	Alexandre Her	569453	19/10/2016 00:00:00	07:15:00	12:51:00	5:36	5,60	Desenvolvimento	513 - Projeto FIDC - Migração ASIS
15	Alexandre Her	569453	19/10/2016 00:00:00	14:05:00	16:31:00	2:26	2,43	Desenvolvimento	513 - Projeto FIDC - Migração ASIS

Figura 36- Relatório de apontamento de horas

Quadro 3 - Refinamento dos indicadores

Código	**Indicador**	**Cálculo**
CTMM	Custo Total Migração Manual	CMM * EMM
CTC	Custo Total Compilador	CC * EC

3.7.1.1 Cálculo viabilidade

Nesta etapa serão feitas duas abordagens de viabilidade, uma por linhas de código (LOC) e outra por pontos de função (FPA). A primeira abordagem dará uma visão baseada no esforço homem/hora de migração. A segunda dará a visão em termos de valor agregado uma vez que o FPA não leva em consideração o tamanho do programa gerado, mas o que é percebido pelo usuário.

Desta forma a equação 1 e equação 2 vão responder se o desenvolvimento do processo automatizado foi viável. Os indicadores utilizados na equação fazem parte do Quadro 2 e do Quadro 3.

$$TLOC\frac{CTMM}{LOC} > CTC + \sum(EMA \times TINM) \times CMM$$

Equação 1 - Fórmula de viabilidade por linhas de código

Os indicadores TLOC e TINM foram obtidos de uma análise quantitativa de todos os códigos fonte que poderão ser migrados. Uma fez a análise contando a quantidade de linhas de cada um dos programas e contando todos os recursos identificados como não convertidos na etapa Conversão Automatizada.

$$TFPA\frac{CTMM}{FPA} > CTC + \sum(EMA \times TINM) \times CMM$$

Equação 2 - Fórmula de viabilidade por FPA

Os indicadores de gestão foram obtidos respectivos sistemas onde estavam armazenados o quadro 4 demostra os valores obtidos. O esforço de migração manual (EMM) é referente a uma migração manual de um dos programas. O programa “ariscdia” foi migrado manualmente e os dados computados são referentes a 428 horas de esforço. Este trabalho manual será utilizado com base para os cálculos relacionados a custo total de migração manual, seja por linha de código ou por análise de ponto de função. A quantidade de linhas de código do programa “ariscdia” foi computado na etapa Conversão Automatizada desta pesquisa. O programa original possui 98.291 linhas de código e foi dimensionado em 80 pontos de função pelos analistas durante fase de planejamento desta conversão manual.

Quadro 4 - Valor dos indicadores coletados dos sistemas de gestão

Indicador	Valor
EC	2.430
CC	R$ 110,00
EMM	428
CMM	R$ 70,00
CMA	R$ 70,00
TLOC	8.460.600
TFPA	1770

Foi feita análise no código fonte dos 100 programas eleitos para conversão em busca de construções que não serão convertidas. Baseado na quantidade encontrada de cada item, é possível, usando a equação 1 calcular a viabilidade da construção do compilador.

Tabela 4 – Cálculo do esforço necessário de intervenção manual

Item	**Quantidade (TINM)**	**EMA**	**EMA x TINM**
Commit	19276	0,01	192,76
Rollback	133	0,01	1,33
Connect	608	0,01	6,08
Disconnect	301	0,01	3,01
Execute Immediate	6112	0,05	305,60
Declare Cursor	1133	0,10	113,30
Open Cursor	1124	0,01	11,24
Close Cursor	1139	0,01	11,39
Fetch	1338	0,01	13,38
Prepare	819	0,05	40,95
OleObject.ConnectToNewObject	205	0,50	102,50
SyntaxFromSQL	727	0,30	218,10
DataStore.Describe	7354	0,30	2.206,20
DataStore.Modify	1216	0,30	364,80
DataStore (DotNotation)	1067	0,30	320,10
Post Function	1198	0,30	359,40
		$\sum(EMA \times TINM) = 4.270,14$	

No fim da

tabela 4 é exibido o cálculo do somatório do de EMA x TINM Equação 1. O esforço total calculado para ajustar todas os itens não migrados foi estimado em 4.270,14 (quatro mil duzentos e setenta virgula quatorze horas) que a um custo de R$ 70,00 (setenta reais) por homem/hora temos um valor total estimado de R$ 298.909,80 (duzentos e noventa e oito mil novecentos e nove reais e oitenta centavos) que deve ser somado com o custo da construção do compilador (CTC) de R$ 267.300,00 (duzentos e sessenta e sete mil e trezentos reais). Este valor deve ser menor que o custo total estimado da migração manual, para que seja economicamente viável a construção do compilador. A figura 37 demonstra o detalhamento do cálculo de viabilidade utilizando linhas de código. A divisão exposta na equação 1 de CTMM por LOC determina o custo da migração manual por linha de código, sendo R$ 29.960,00 (vinte e nove mil novecentos e sessenta reais) o custo da migração manual do programa “ariscdia” dividido pela quantidade de linhas do programa, 98.291 (noventa e oito mil duzentos e noventa e um). Este valor é multiplicado pelo valor de todas as linhas de código pendentes de migração, no caso 8.460.600 (oito milhões quatrocentos e sessenta mil e seiscentos).

$$8.460.600\frac{29.960}{98291} > 267.300 + 298.909{,}80$$

Figura 37- Desmembramento do cálculo de viabilidade por linhas de código

O custo total estimado para a migração manual de todos os programas está estimado em R$ 2.578.868,62 (dois milhões quinhentos e setenta e oito mil oitocentos e sessenta e oito reais e sessenta e dois centavos) contra um custo total de R$ 566.209,80 (quinhentos e sessenta e sei mil duzentos e nove reais e oitenta centavos) da construção do compilador mais a estimativa do de intervenção manual após a migração automatizada.

Esta análise também pode ser feita por contagem de pontos de função, no caso ao invés de calcular o custo da migração de uma linha de código calcula-se o custo de um ponto de função. Neste caso troca-se apenas a divisão de CTMM por LOC por CTMM por FPA

(contagem de pontos de função do programa “ariscdia”) e multiplica-se pela quantidade de pontos de função de todos os programas pendentes de migração. Devido à falta de acesso à informações não foi possível efetuar o cálculo da viabilidade com FPA.

4 Resultados

Nesta seção vamos discutir os resultados da pesquisa. As etapas da pesquisa serão apresentadas com seus devidos resultados consolidados, já relatados anteriormente. Esta seção tem como objetivo simplificar a análise dos resultados da pesquisa em um único ponto deste documento.

4.1.1 Construção do compilador

Durante a construção do compilador foram coletadas informações de apontamento de horas (quantidade de homem/hora gastos) esta informação foi crucial para o cálculo da viabilidade apresentado na seção 3.7 - Avaliação da viabilidade da automação. Foram gastas 2.430 (duas mil quatrocentas e trinta) horas a um custo de R$ 110,00 (cento e dez reais) homem/hora. Assim o custo total do desenvolvimento do compilador foi de R$ 267.300,00 (duzentos e sessenta e sete mil e trezentos reais).

4.1.2 Conversão Automatizada

Foram convertidos dois programas do Sistema de Gestão de Fundos de Recebíveis, “remcli” e “sfrintsac”. Após a conversão pode observar um percentual geral de linhas convertidas satisfatório. Foram convertidas 221.016 linhas de código o que representa 99,4% de linhas de código convertidas.

Os erros de compilação representaram 0,2% (dois décimos percentuais) da quantidade total de linhas de código do fonte original. E podem ser desprezados neste caso. A maioria dos erros são referentes ao sistema de tipos, como: comparação entre *strings* utilizando operadores relacionais, alguns erros de tipagem onde o analisador semântico calculou o tipo de alguma expressão equivocadamente e objetos referenciados no código, porém não migrados. Estes problemas serão tratados como defeitos do compilador e serão corrigidos no futuro, mas foram registrados nos resultados desta pesquisa.

Quadro 5 - Funcionalidades não migradas

Funcionalidade não migrada	**Quantidade de aparições**
Commit	459
Rollback	2
Connect	10
Disconnect	5
Execute Immediate	95
Declare Cursor	18
Open Cursor	17
Close Cursor	19
Fetch	21
Prepare	12
OleObject.ConnectToNewObject	4
SyntaxFromSQL	26
DataStore.Describe	113
DataStore.Modify	19
DataStore (DotNotation)	14
Post Function	19
853 itens não migrados	

Os principais itens não convertidos estão ligados a acessos alternativos ao banco de dados via cursores, manipulação de transações.

4.1.3 Workshop

O Workshop foi executado em duas etapas: a aplicação de um questionário e um grupo focal.

O questionário aplicado tinha como objetivo a avaliação da qualidade do código fonte gerado. Foram apresentados 10 trechos de código originais e suas respectivas versões convertidas. As perguntas podem ser vistas no ANEXO I deste documento.

Os resultados do questionário indicam uma boa aceitação, por parte dos programadores, do código gerado. Para chegar ao resultado levou-se em consideração a moda das respostas. O Quadro 7 demostra em todos os questionamentos o resultado foi concordo totalmente.

Quadro 6-Resultado analítico da pesquisa de qualidade do fonte migrado.

Itens	**Respostas**				
	Não concordo totalmente	**Não concordo parcialmente**	**Indiferente**	**Concordo parcialmente**	**Concordo totalmente**
Este código após a migração está legível	3	3	3	27	204
Este código após a migração pode ser mantido	0	0	0	9	231
Este código após a migração pode ser evoluído	0	3	3	21	213
Este código após migração está escrito de forma que não tenha perda de desempenho em relação ao original	0	3	15	15	207
Este código após a migração manteve a semântica original	0	3	0	9	228

Quadro 7 - Resultado consolidado da pesquisa de qualidade do fonte migrado

Itens	**Resultado**
Este código após a migração está legível	Concordo totalmente
Este código após a migração pode ser mantido	Concordo totalmente
Este código após a migração pode ser evoluído	Concordo totalmente
Este código após migração está escrito de forma que não tenha perda de desempenho em relação ao original	Concordo totalmente
Este código após a migração manteve a semântica original	Concordo totalmente

O grupo focal teve como objetivo levantar o esforço necessário para adaptar os itens não convertidos pelo processo automatizado. Os esforços forma levantados em homem/hora de forma que pudessem ser utilizados para análise da viabilidade.

Tabela 5 - Esforço para intervenção manual de itens não migrados

Construção	**Esforço (h/h)**
Commit	0,01
Rollback	0,01
Connect	0,01
Disconnect	0,01
Execute Immediate	0,05
Declare Cursor	0,10
Open Cursor	0,01
Close Cursor	0,01
Fetch	0,01
Prepare	0,05
OleObject.ConnectToNewObject	0,50
SyntaxFromSQL	0,30
DataStore.Describe	0,30
DataStore.Modify	0,30
DataStore (DotNotation)	0,30
Post Function	0,30
Total: 16 itens	

A tabela 5 demonstra o resultado da avaliação feita pelos programadores com o esforço necessário para intervir manualmente para uma única ocorrência de cada tipo de item não migrado. Estas informações foram utilizadas na seção Avaliação da viabilidade da automação.

4.1.4 Síntese dos desafios

Nesta etapa foram levantados os principais desafios com suas respectivas soluções. Foram registrados e documentados levantados 27 desafios. Cujo destes apenas 6 se aplicam apenas à conversões de PowerBuilder para C#. A síntese dos desafios e suas soluções permitem que o experimento possa ser aplicado com maior facilidade para outros casos. O

Quadro 8 descreve outras linguagens onde os desafios catalogados também terão que ser solucionados.

Quadro 8 – Relação de desafios com outras linguagens origem

Desafio	**Visual Basic**	**Javascript**	**PL/SQL**	**Transact-SQL**	**COBOL**	**C**	**C++**
Case Insensitive	x		x	x	x		
Keywords and Identifiers Ambiguity							
Line Continuing	x						
Line Comments			x	x			
Choose case	x						
Embedded SQL			x	x			
1 Based Array	x						
Dynamic Allocated Arrays		x					
Function Call	x		x	x	x	x	
Initial Value of Variables	x	x	x	x	x		
Primitive Type Date	x	x	x	x	x		
Primitive Type Time			x	x	x		
File Manipulation	x				x	x	x
Global System Functions	x		x	x	x	x	
Global User Functions	x		x	x	x	x	
Datastore Class							
Datawindow Objects							
Transaction Class							
Case Insensitive (Transformação)	x		x	x	x		
Automatic Number Casting	x	x	x	x			
Char and String Literals	x		x	x			
Automatic Casting	x	x					
Default Acces Modifiers							
Native Functions	x		x	x	x	x	x
Global Variables	x				x	x	
Exponential Operator	x				x		
Integer Precision							

4.1.5 Avaliação da viabilidade da automação

A análise da viabilidade da automação permitiu verificar para um caso específico quando é viável a construção de um compilador para executar a migração de código fonte. Para o experimento aplicado a construção foi economicamente viável. O custo total previsto de migração de todos os programas R$ 2.578.868,62 (dois milhões quinhentos e setenta e oito mil oitocentos e sessenta e oito reais e sessenta e dois centavos) é muito superior aos R$ 566.209,80 (quinhentos e sessenta e sei mil duzentos e nove reais e oitenta centavos) referentes a construção do compilador mais as estimativas de intervenção manual.

5 Conclusão

A presente pesquisa aplicou um processo automatizado em uma empresa de software visando avaliar a viabilidade técnica e econômica do mesmo. Foi desenvolvido um compilador que migra códigos escritos na plataforma de desenvolvimento PowerBuilder para C# da plataforma .Net.

Durante a construção do compilador foi possível catalogar, de forma a contribuir para trabalhos futuros e aumentar o conhecimento na área, os principais desafios encontrados na literatura e durante a aplicação da pesquisa, assim como as soluções propostas para cada um. O compilador conseguiu índices superiores a 99% (noventa e nove por cento) de conversão de linhas de código índices também conseguidos por NEWCOMB & DOBLAR (2001) em seu trabalho sobre a transformação automatizada de códigos legados.

Uma das premissas do projeto do compilador era que o código gerado seria patrimônio da empresa onde a pesquisa foi executada, tendo necessidade de continuar a ser mantido pro programadores na linguagem alvo, neste caso C#. O resultado da pesquisa demonstrou um alto índice de aceitação do nível de qualidade do código gerado, podendo este ser mantido e evoluído e continuar sendo um ativo da empresa com as mesmas qualidades do código fonte original, porém em uma plataforma mais moderna com mais compatível com hardware e software mais modernos e com potencial evolutivo.

Em um universo de mais de oito milhões de linhas de código e obtido o surpreendente índice de superior a 99% (noventa e nove por cento) de conversão um por cento representa cerca de 80.000 (oitenta mil) linhas de código não convertidos e que necessitam de intervenção manual. Nesta pesquisa foi utilizada técnica de grupo focal para obter estimativas das construções não convertidas pelo compilador. Foi possível obter estimativas individuais de cada construção e estimar, através de contagem destes itens individualmente existentes em todos os códigos fonte com potencial para migração automatizada.

NEWCOMB & DOBLAR (2001) fizeram uma análise parecida utilizando dados obtidos através do Gartner Group a respeito da produtividade de programadores durante o processo manual de conversão de código. A métrica utilizada é de 160 (cento e sessenta) linhas de código por dia. Utilizando estas métricas seriam, considerando trabalho diário de oito horas, necessários cerca de 4.000 mil horas de trabalho o que surpreendentemente está bem próximo dos números obtidos nesta pesquisa.

A pesquisa demonstrou a viabilidade econômica do processo. Pode parecer óbvio quando comparamos um processo manual contra um processo automatizado qual deles será o mais viável. Porém quando não se tem o ferramental *a priori* e este necessita ser construído deve-se avaliar a viabilidade da construção deste ferramental e validar contra todo o estoque de programas a serem convertidos. Neste caso o processo automatizado se demonstrou economicamente viável o investimento da construção do ferramental, neste caso o compilador é pago ao converter cerca de 20% (vinte por cento) do estoque de programas.

5.1 Limitações da Pesquisa

Neste trabalho optou-se por trabalhar com as linguagens PowerScript (linguagem da plataforma PowerBuilder) como linguagem fonte e C# como linguagem alvo. Os desafios e dados obtidos nesta pesquisa se limitam ao processo envolvendo estas linguagens.

Os itens citados na seção Síntese dos desafios, se aplicam em grande parte na relação entre a plataforma PowerBuilder e a plataforma .Net e entre as linguagens PowerScript e C#. Apesar de vários desafios enfrentados e catalogados serem pertinentes a outras relações linguagem-plataforma outros não foram identificados por esta pesquisa que se limitou a apenas uma relação.

A segunda limitação é referente ao local de aplicação da pesquisa. Ela foi aplicada em apenas uma empresa de desenvolvimento de software e as informações de produtividade, qualidade e interpretação do código gerado se limitam à cultura e experiência proporcionada

por esta organização aos seus colaboradores. Portanto, informações de produtividade de acerto manual e migração manual são particulares e para aplicação em outras organizações devem ser medidos novamente.

A terceira limitação é referente ao tipo de construções inicialmente planejados para esta pesquisa. O compilador limita-se a converter códigos que não trabalhem com interface gráfica. O que limita de programas com potencial para migração assim limitando os desafios encontrados e análise de complexidade o que impacta diretamente a viabilidade econômica.

Outra limitação é o fato desta pesquisa ter como um de seus objetivos a avaliação da qualidade do código gerado, porém não avalia a qualidade do programa final. Assim mesmo tendo qualidade na geração do código o programa final ode se comportar de forma inesperada e não foi avaliado por esta pesquisa.

Contudo, apesar das limitações, acredita-se que a metodologia utilizada nesta pesquisa pode ser replicada para outras linguagens em outras organizações e contribuir para que trabalhos mais amplos possam ser feitos.

5.2 Trabalhos futuros

Como trabalhos futuros, vislumbra-se uma pesquisa mais ampla tendo como o foco a migração não apenas de linguagens e plataformas, mas também de paradigmas. A tecnologia evolui rapidamente sistemas que são executados em computadores pessoais precisam ser migrados para tablets e smartphones.

Pretende-se também a replicação da pesquisa em forma de artigo envolvendo outras linguagens e outras organizações de culturas diferentes e comparar os resultados com este trabalho.

Outro ponto a ser analisado refere-se à qualidade do programa final. Esta pesquisa se limitou a análise de código fonte e não expandiu a análise até o programa gerado. Avaliar o a

qualidade do programa gerado contabilizando *bugs* e esforço de retrabalho para correções do programa alvo são pontos a serem observados em trabalhos futuros.

Existe muito ainda a se fazer nesta área. Todo software tende a ficar desatualizado em ralação às variáveis externas como hardware, softwares e paradigmas de desenvolvimento como *Web*, *Mobile* e *cloud computing*. Ainda há necessidade de muita pesquisa nesta área pois a tecnologia avança rapidamente criando novos desafios a serem descobertos catalogados e solucionados.

ANEXO I

Original

```
long    i
string  ls_concat[], ls_vazio[]

do while il_size > 1
    for i=1 to il_size/2
        ls_concat[i] = is_concat[2*i -1] + is_concat[2*i]
    next
    if mod(il_size, 2) = 1 then
        ls_concat[il_size/2+1] = is_concat[il_size]
    end if
    is_concat = ls_concat
    ls_concat = ls_vazio
    il_size = il_size/2 + mod(il_size,2)
loop

return is_concat[1]
```

Convertido

```
int i = default(int);
PBArray<string> ls_concat = default(string[]), ls_vazio = default(string[]);
while (il_size > 1)
{
    for (i = 1; i <= il_size / 2; i++)
    {
        ls_concat[i - 1] = is_concat[(2 * (short)i - 1) - 1] + is_concat[(2 * (short)i) - 1];
    }

    if (mod(il_size, 2) == 1)
    {
        ls_concat[(il_size / 2 + 1) - 1] = is_concat[il_size - 1];
    }

    is_concat = ls_concat;
    ls_concat = ls_vazio;
    il_size = il_size / 2 + (int)mod(il_size, 2);
}

return is_concat[0];
```

Original

```
if (upper(trim(mid(sqlca.dbms,5,6))) = 'ORACLE' or upper(trim(mid(sqlca.dbms,11,6))) = 'ORACLE') then
    ls_decimais = fill('0', al_qtd_decimais)
    ls_inteiros = fill('0', al_tamanho - (al_qtd_decimais))

    choose case as_separador
        case 'N'
            ls_query = "TRIM(REPLACE(REPLACE(TO_CHAR(NVL(" + as_campo + ", 0), '" + ls_inteiros + "D" + ls_decimais + "'), '.', ''), ',', ''))"

        case ','
            ls_query = "TRIM(REPLACE(TO_CHAR(NVL(" + as_campo + ", 0), '" + ls_inteiros + "D" + ls_decimais + "'), '.', ','))"

        case '.'
            ls_query = "TRIM(REPLACE(TO_CHAR(NVL(" + as_campo + ", 0), '" + ls_inteiros + "D" + ls_decimais + "'), ',', '.'))"

        case else
            ls_query = "TRIM(REPLACE(REPLACE(TO_CHAR(NVL(" + as_campo + ", 0), '" + ls_inteiros + "D" + ls_decimais + "'), '.', ''), ',', ''))"
    end choose
else
```

Convertido

```
if ((Globals.sqlca.DBMS.ToUpper().Trim().Substring(5, 6) == "ORACLE" || Globals.sqlca.DBMS.ToUpper().Trim().Substring(11, 6) == "ORACLE"))
{
    ls_decimais = new string('0', al_qtd_decimais);
    ls_inteiros = new string('0', al_tamanho - (al_qtd_decimais));
    if (as_separador == "N")
    {
        ls_query = "TRIM(REPLACE(REPLACE(TO_CHAR(NVL(" + as_campo + ", 0), '" + ls_inteiros + "D" + ls_decimais + "'), '.', ''), ',', ''))";
    }
    else if (as_separador == ",")
    {
        ls_query = "TRIM(REPLACE(TO_CHAR(NVL(" + as_campo + ", 0), '" + ls_inteiros + "D" + ls_decimais + "'), '.', ','))";
    }
    else if (as_separador == ".")
    {
        ls_query = "TRIM(REPLACE(TO_CHAR(NVL(" + as_campo + ", 0), '" + ls_inteiros + "D" + ls_decimais + "'), ',', '.'))";
    }
    else
    {
        ls_query = "TRIM(REPLACE(REPLACE(TO_CHAR(NVL(" + as_campo + ", 0), '" + ls_inteiros + "D" + ls_decimais + "'), '.', ''), ',', ''))";
    }
}
else
```

Original

```
datetime ldt_hdrdathor
string ls_retorno
str_headertabela lstr_header
integer li_ret

//Busca as informações necessárias para montar o header das tabelas
u_header_tabelas luo_reader_tabelas
try
    luo_reader_tabelas = create u_header_tabelas

    li_ret = luo_reader_tabelas.uf_header_tabelas( lstr_header)


    as_hdrdathor = lstr_header.dataalteracao
    as_hdrcodusu = lstr_header.usuario
    as_hdrcodetc = lstr_header.estacao
    as_hdrcodpgr = lstr_header.programa

finally
    destroy luo_reader_tabelas
end try

return ''
```

Convertido

```
DateTime ldt_hdrdathor = default (DateTime);
string ls_retorno = default (string);
str_headertabela lstr_header = default (str_headertabela);
short li_ret = default (short);
//Busca as informações necessárias para montar o header das tabelas
u_header_tabelas luo_reader_tabelas = default (u_header_tabelas);
try
{
    luo_reader_tabelas = new u_header_tabelas();
    li_ret = luo_reader_tabelas.uf_header_tabelas(ref lstr_header);
    as_hdrdathor = lstr_header.dataalteracao;
    as_hdrcodusu = lstr_header.usuario;
    as_hdrcodetc = lstr_header.estacao;
    as_hdrcodpgr = lstr_header.programa;
}
finally
{
    luo_reader_tabelas = null;
}

return "";
```

Original

```
ll_NumLinhas = lds_VersaoApl.RowCount()

IF ll_NumLinhas > 0 THEN
    is_hdrdathor_apl = lds_VersaoApl.GetItemString(1, 'hdrdathor')
    is_hdrcodusu_apl = lds_VersaoApl.GetItemString(1, 'hdrcodusu')
    is_hdrcodetc_apl = lds_VersaoApl.GetItemString(1, 'hdrcodetc')
    is_hdrcodpgr_apl = lds_VersaoApl.GetItemString(1, 'hdrcodpgr')
    is_cvadesmodulo  = lds_VersaoApl.GetItemString(1, 'cvadesmodulo')
    is_cvavrsbdreq   = lds_VersaoApl.GetItemString(1, 'cvavrsbdreq')
    is_cvavrsprog    = lds_VersaoApl.GetItemString(1, 'cvavrsprog')

    IF Not IsNull(idt_cvadatbloqueio) Then
        ldt_cvadatbloqueio_bd = lds_VersaoApl.GetItemDateTime(1, 'cvadatbloqueio')
        IF Not IsNull(ldt_cvadatbloqueio_bd) THEN
            IF ldt_cvadatbloqueio_bd > idt_cvadatbloqueio THEN
                ls_Retorno = "Foi disponibilizada e executada neste ambiente, uma versão posterior a esta em execução.~r" + &
                            " (Versão em execução: " + is_cvavrsprog_inf + " Data: " + String(idt_cvadatbloqueio,"dd/mm/yyyy hh:mm:ss") + ")~r" +
                            " (Última versão disponibilizada: " + is_cvavrsprog + " Data: " + String(ldt_cvadatbloqueio_bd,"dd/mm/yyyy hh:mm:ss"
            END IF
        END IF
    END IF

ELSE
    IF Not IsNull(idt_cvadatbloqueio) THEN
        is_cvavrsprog = ls_VersaoApl
        ls_Retorno = ''
    ELSE
        ls_Retorno = 'Erro ao buscar a versão da aplicação, favor entrar em contato com o responsável pelo sistema.~rd_versaoapl'
    END IF
END IF
```

Convertido

```
ll_NumLinhas = lds_VersaoApl.rowcount();
if (ll_NumLinhas > 0)
{
    is_hdrdathor_apl = lds_VersaoApl.getitemstring(1, "hdrdathor");
    is_hdrcodusu_apl = lds_VersaoApl.getitemstring(1, "hdrcodusu");
    is_hdrcodetc_apl = lds_VersaoApl.getitemstring(1, "hdrcodetc");
    is_hdrcodpgr_apl = lds_VersaoApl.getitemstring(1, "hdrcodpgr");
    is_cvadesmodulo = lds_VersaoApl.getitemstring(1, "cvadesmodulo");
    is_cvavrsbdreq = lds_VersaoApl.getitemstring(1, "cvavrsbdreq");
    is_cvavrsprog = lds_VersaoApl.getitemstring(1, "cvavrsprog");
    if (!(idt_cvadatbloqueio == null))
    {
        ldt_cvadatbloqueio_bd = lds_VersaoApl.getitemdatetime(1, "cvadatbloqueio");
        if (!(isnull(ldt_cvadatbloqueio_bd)))
        {
            if (ldt_cvadatbloqueio_bd > idt_cvadatbloqueio)
            {
                ls_Retorno = "Foi disponibilizada e executada neste ambiente, uma versão posterior a esta em execução.\\r" + " (\
            }
        }
    }
}
else if (!(isnull(idt_cvadatbloqueio)))
{
    is_cvavrsprog = ls_VersaoApl;
    ls_Retorno = "";
}
else
{
    ls_Retorno = "Erro ao buscar a versão da aplicação, favor entrar em contato com o responsável pelo sistema.\\rd_versaoapl";
}
```

Original

```
//Dados da Versão
string gs_data_versao, gs_versao_sis, gs_versao_build, gs_versao_banco, gs_data_bloqueio


long      gl_emp, gl_banco
datetime  gdt_data_atual, gdt_datahora_serv
string gs_codusu, gs_codetc, gs_codpgr, gs_path, gs_param

//Variaveis para utilizacao do manual de mensagens
string gs_arquivoini, gs_queue
integer   gi_tip_arquivo
double    gdb_retorno = 1, gdb_seq_rotina
udr_serv_queue    gu_queue

u_dr_lock     dr_lock
gtaeglctrver istr_versao

s_parms gstr_arg

// arquivo de debug
long    gl_arq_debug
string gs_debug

// Originador
string gs_nome_cedente = 'Originador', gs_nome_cedente_plural = 'Originadores', gs_nome_cedente_abrev = 'Orig.'

// Fundo
string gs_nome_fundo = 'Fundo', gs_nome_fundo_plural = 'Fundos', gs_nome_fundo_abrev = 'Fdo.'

// Nome do sistema
string gs_nome_sis = 'Sistema de Controle de Cessão'

// trata feito conferido
string gs_feitoconferido

// Conexão
transaction gt_conexao
```

Convertido

```
public static class Globals
{
    public static string gs_data_versao;
    public static string gs_versao_sis;
    public static string gs_versao_build;
    public static string gs_versao_banco;
    public static string gs_data_bloqueio;
    public static int gl_emp;
    public static int gl_banco;
    public static DateTime gdt_data_atual;
    public static DateTime gdt_datahora_serv;
    public static string gs_codusu;
    public static string gs_codetc;
    public static string gs_codpgr;
    public static string gs_path;
    public static string gs_param;
    public static string gs_arquivoini;
    public static string gs_queue;
    public static short gi_tip_arquivo;
    public static double gdb_retorno = 1;
    public static double gdb_seq_rotina;
    public static udr_serv_queue gu_queue;
    public static u_dr_lock dr_lock;
    public static gtaeglctrver istr_versao;
    public static s_parms gstr_arg;
    public static int gl_arq_debug;
    public static string gs_debug;
    public static string gs_nome_cedente = "Originador";
    public static string gs_nome_cedente_plural = "Originadores";
    public static string gs_nome_cedente_abrev = "Orig.";
    public static string gs_nome_fundo = "Fundo";
    public static string gs_nome_fundo_plural = "Fundos";
    public static string gs_nome_fundo_abrev = "Fdo.";
    public static string gs_nome_sis = "Sistema de Controle de Cessão";
    public static string gs_feitoconferido;
    public static Transaction gt_conexao;
```

Original

```
long ll_count, ll_num_dias, ll_num_fim_semana

// Número de dias corridos
ll_num_dias = daysafter(dataini, datafim)

// Número de finais de semana entre as datas
ll_num_fim_semana = Truncate(ll_num_dias / 7, 0)

// Se o dia da semana da data inicial for maior que o da data final e a data final não for final de semana
if daynumber(dataini) > daynumber(datafim) AND daynumber(datafim) <> 7 AND daynumber(datafim) <> 1 then
    ll_num_fim_semana++
end if

// Retira sabado e domingo da diferença de dias
ll_num_dias = ll_num_dias - ( ll_num_fim_semana * 2)

// Se data final é um sábado
if daynumber(datafim) = 7 then
    ll_num_dias = ll_num_dias - 1
end if

// Se data final é um domingo
if daynumber(datafim) = 1 then
    ll_num_dias = ll_num_dias - 2
end if
```

Convertido

```
int ll_count = default (int), ll_num_dias = default (int), ll_num_fim_semana = default (int);
// Número de dias corridos
ll_num_dias = daysafter(dataini, datafim);
// Número de finais de semana entre as datas
ll_num_fim_semana = (int)truncate(ll_num_dias / 7, 0);
// Se o dia da semana da data inicial for maior que o da data final e a data final não for final de semana
if (daynumber(dataini) > daynumber(datafim) && daynumber(datafim) != 7 && daynumber(datafim) != 1)
{
    ll_num_fim_semana++;
}

// Retira sabado e domingo da diferença de dias
ll_num_dias = ll_num_dias - (ll_num_fim_semana * 2);
// Se data final é um sábado
if (daynumber(datafim) == 7)
{
    ll_num_dias = ll_num_dias - 1;
}

// Se data final é um domingo
if (daynumber(datafim) == 1)
{
    ll_num_dias = ll_num_dias - 2;
}
```

Original

```
string ls_codsisdll

Setnull(is_hdrdathor_bd)
Setnull(is_hdrcodusu_bd)
Setnull(is_hdrcodetc_bd)
Setnull(is_hdrcodpgr_bd)
Setnull(is_cvbvrsbd)

ls_codsisdll = is_codsis + 'DDL'

if ib_controlesequencial then

    SELECT cvb.hdrdathor,
           cvb.hdrcodusu,
           cvb.hdrcodetc,
           cvb.hdrcodpgr,
           cvb.cvbvrsbd
     INTO :is_hdrdathor_bd,
           :is_hdrcodusu_bd,
           :is_hdrcodetc_bd,
           :is_hdrcodpgr_bd,
           :is_cvbvrsbd
    FROM ugetblcvb cvb
    INNER JOIN (SELECT max(hdrdathor) hdrdathor, max(a.cvbvrsbd) cvbvrsbd
                FROM ugetblcvb a
               WHERE a.cvbnomlinhasis = :is_nomlinhasis
                 AND a.cvbcodsis = :ls_codsisdll
                 AND a.cvbvrsbd = (SELECT MAX(cvbvrsbd)
                                     FROM ugetblcvb b
                                    WHERE b.cvbnomlinhasis = a.cvbnomlinhasis
                                      AND b.cvbcodsis = a.cvbcodsis
                                      AND b.cvbvrsbd like 'V%' ) ) aux
       on aux.hdrdathor = cvb.hdrdathor and aux.cvbvrsbd = cvb.cvbvrsbd
    WHERE cvb.cvbnomlinhasis = :is_nomlinhasis
      AND cvb.cvbcodsis = :ls_codsisdll
      AND cvb.cvbvrsbd like 'V%';
```

Convertido

```
        string ls_codsisdll = default(string);
        is_hdrdathor_bd = null;
        is_hdrcodusu_bd = null;
        is_hdrcodetc_bd = null;
        is_hdrcodpgr_bd = null;
        is_cvbvrsbd = null;
        ls_codsisdll = is_codsis + "DDL";
        if (ib_controlesequencial)
        {
            Globals.sqlca.Select(@"SELECT cvb.hdrdathor,
        cvb.hdrcodusu,
        cvb.hdrcodetc,
        cvb.hdrcodpgr,
        cvb.cvbvrsbd

 FROM ugetblcvb cvb
INNER JOIN (SELECT max(hdrdathor) hdrdathor, max(a.cvbvrsbd) cvbvrsbd
                  FROM ugetblcvb a
                 WHERE a.cvbnomlinhasis = :v1
                   AND a.cvbcodsis = :v2
                   AND a.cvbvrsbd = (SELECT MAX(cvbvrsbd)
                                          FROM ugetblcvb b
                                         WHERE b.cvbnomlinhasis = a.cvbnomlinhasis
                                           AND b.cvbcodsis = a.cvbcodsis
                                           AND b.cvbvrsbd like 'V%' ) ) aux
    on aux.hdrdathor = cvb.hdrdathor and aux.cvbvrsbd = cvb.cvbvrsbd
 WHERE cvb.cvbnomlinhasis = :v3
   AND cvb.cvbcodsis = :v4
   AND cvb.cvbvrsbd like 'V%'", is_nomlinhasis, ls_codsisdll, is_nomlinhasis, ls_codsisdll).GetResult(r =>
            {
                is_hdrdathor_bd = r.GetString(0);
                is_hdrcodusu_bd = r.GetString(1);
                is_hdrcodetc_bd = r.GetString(2);
                is_hdrcodpgr_bd = r.GetString(3);
                is_cvbvrsbd = r.GetString(4);
            }
```

Original

st_cpf (sfrintsac) (D:\TFS\41\Ferramenta\Parser\Trunk\Main\Conversor.Test\Cases\FIDC_Batch\Main\sfr\intout\st_sfr_bd.pbl) - Structure

Type	Variable Name
string	cpfcodsit
string	cpfcodtip
string	cpfidcreexe
string	fdoidccnpj
string	cpfdesproc
string	cpfdespath
string	cpfdespathrel
string	cpfdesusu
string	cpfdesemail
datetime	cpfdatatusis
datetime	cpfdatatual
datetime	cpfdatante
datetime	cpfdatpos
long	fdocodemp
long	cpfcodproc
long	cpfcodctaproc
long	cpfnumprocant
long	cpfnumcliclo
long	

Convertido

```
public struct st_cpf
{
    public string cpfcodsit;
    public string cpfcodtip;
    public string cpfidcreexe;
    public string fdoidccnpj;
    public string cpfdesproc;
    public string cpfdespath;
    public string cpfdespathrel;
    public string cpfdesusu;
    public string cpfdesemail;
    public DateTime cpfdatatusis;
    public DateTime cpfdatatual;
    public DateTime cpfdatante;
    public DateTime cpfdatpos;
    public int fdocodemp;
    public int cpfcodproc;
    public int cpfcodctaproc;
    public int cpfnumprocant;
    public int cpfnumcliclo;
}
```

Original

```
// Busca usuário logado
string ls_null, ls_tam
uint  lui_tam
long ll_tam
constant int ci_max = 30

Throwable ocorreu_erro

If ib_inicializou then
    as_user_name = is_user_name
else
    Try
        as_user_name = space(255)
        lui_tam = 255
        lui_tam = Wnetgetuser(ls_null, as_user_name, lui_tam)

        if lui_tam <> 0 then
            throw ocorreu_erro
        end if

        if LenA(as_user_name) > ci_max then
            as_user_name = LeftA(as_user_name, ci_max -1) + '*'
        end if

        is_user_name = as_user_name
    catch (throwable ex)
        as_user_name = ''
        is_user_name = ''
        is_erro = "User name não identificado."
        return -1
    end try
end if

return 1
```

Convertido

```
// Busca usuário logado
string ls_null = default(string), ls_tam = default(string);
ushort lui_tam = default(ushort);
int ll_tam = default(int);
const short ci_max = 30;
throwable ocorreu_erro = default(throwable);
if (ib_inicializou)
{
    as_user_name = is_user_name;
}
else
{
    try
    {
        as_user_name = space(255);
        lui_tam = (ushort)255;
        lui_tam = (ushort)WNetGetUser(ref ls_null, ref as_user_name, ref lui_tam);
        if (lui_tam != (ushort)0)
        {
            throw ocorreu_erro;
        }

        if (lena(as_user_name) > ci_max)
        {
            as_user_name = as_user_name.Substring(0, ci_max - 2) + "*";
        }

        is_user_name = as_user_name;
    }
    catch (throwable ex)
    {
        as_user_name = "";
        is_user_name = "";
        is_erro = "User name não identificado.";
        return -1;
    }
}

return 1;
```

Original

```
string ls_aux
st_param param

//Processa tipo de execução
param.s_tipo_proc = f_recupera_argumento("/t",as_comando)

//Recupera a empresa
ls_aux = f_recupera_argumento("/e", as_comando)
if isnumber(ls_aux) then
    param.l_cod_emp = long(ls_aux)
end if

//Recupera o número do range
ls_aux = f_recupera_argumento("/r", as_comando)
if isnumber(ls_aux) then
    param.l_cod_range = long(ls_aux)
end if

//Recupera o contrato inicial
param.s_con_ini = f_recupera_argumento("/ci", as_comando)


//Recupera o contrato final
param.s_con_fim = f_recupera_argumento("/cf", as_comando)

//Recupera caminho do log
param.s_log = f_recupera_argumento("/log",as_comando)

//Recupera caminho do log de erro
param.s_logerro = f_recupera_argumento("/logerro",as_comando)

//Recupera o fundo
param.s_fundo = f_recupera_argumento("/fundo",as_comando)

//Marca se o processamento deve ser feito por range ou não
param.b_processa_por_range = param.s_tipo_proc <> ""

return param
```

Convertido

```
string ls_aux = default (string);
st_param param = default (st_param);
//Processa tipo de execução
param.s_tipo_proc = GlobalFunctions.f_recupera_argumento("/t", as_comando);
//Recupera a empresa
ls_aux = GlobalFunctions.f_recupera_argumento("/e", as_comando);
if (isnumber(ls_aux))
{
    param.l_cod_emp = @long(ls_aux);
}

//Recupera o número do range
ls_aux = GlobalFunctions.f_recupera_argumento("/r", as_comando);
if (isnumber(ls_aux))
{
    param.l_cod_range = @long(ls_aux);
}

//Recupera o contrato inicial
param.s_con_ini = GlobalFunctions.f_recupera_argumento("/ci", as_comando);
//Recupera o contrato final
param.s_con_fim = GlobalFunctions.f_recupera_argumento("/cf", as_comando);
//Recupera caminho do log
param.s_log = GlobalFunctions.f_recupera_argumento("/log", as_comando);
//Recupera caminho do log de erro
param.s_logerro = GlobalFunctions.f_recupera_argumento("/logerro", as_comando);
//Recupera o fundo
param.s_fundo = GlobalFunctions.f_recupera_argumento("/fundo", as_comando);
//Marca se o processamento deve ser feito por range ou não
param.b_processa_por_range = param.s_tipo_proc != "";
return param;
```

ANEXO II

Relação de funções nativas do PowerBuilder com o C#

Função PB	Mapeamento	Função C#
abs	Transformação direta	Math.Abs
acos	Transformação direta	Math.Acos
addtolibrarylist	N/A	
asc	Transformação direta	char.ConvertToUtf32
asca	Transformação indireta	cast para byte
asin	Transformação direta	Math.Asin
atan	Transformação direta	Math.Atan
beep	N/A	
blob	Emulação de Função	
blobedit	N/A	
blobedit	N/A	
blobmid	N/A	
blobmid	N/A	
ceiling	Transformação direta	Math.Ceiling
changedirectory	Transformação indireta	Environment.CurrentDirectory
char	Transformação direta	Convert.ToChar
chara	Transformação direta	Convert.ToChar
choosecolor	N/A	
classname	Emulação de Função	
clipboard	N/A	
close	N/A	
closechannel	N/A	
closewithreturn	N/A	
commandparm	N/A	
cos	Transformação direta	Math.Cos
cpu	Transformação direta	Environment.TickCount
createdirectory	Transformação direta	System.IO.Directory.CreateDirectory
date	Emulação de Função	
datetime	Transformação direta	Convert.ToDateTime
day	Transformação indireta	DateTime.Day ou Date.Day
dayname	Emulação de Função	
daynumber	Emulação de Função	
daysafter	Emulação de Função	
dec	Transformação direta	Convert.ToDecimal
directoryexists	Transformação direta	System.IO.Directory.Exists
doscript	N/A	
double	Transformação direta	Convert.ToDouble
draggedobject	N/A	
execremote	N/A	
exp	Transformação direta	Math.Exp

fact	Emulação de Função	
fileclose	Emulação de Função	
filecopy	Transformação direta	System.IO.File.Copy
filedelete	Transformação direta	System.IO.File.Delete
fileencoding	Emulação de Função	
fileexists	Transformação direta	System.IO.File.Exists
filelength	Emulação de Função	
filelength64	Emulação de Função	
filemove	Transformação direta	System.IO.File.Move
fileopen	Emulação de Função	
fileread	Emulação de Função	
filereadex	Emulação de Função	
fileseek	Emulação de Função	
fileseek64	Emulação de Função	
filewrite	Emulação de Função	
filewriteex	Emulação de Função	
fill	Transformação direta	new string(' ', length)
filla	Transformação direta	new string(' ', length)
fillw	Transformação direta	new string(' ', length)
findclassdefinition	N/A	
findfunctiondefinition	N/A	
findtypedefinition	N/A	
fromansi	Emulação de Função	
fromunicode	Emulação de Função	
fromutf8	Emulação de Função	
garbagecollectgettimelimit	N/A	
garbagecollectsettimelimit	N/A	
getapplication	N/A	
getcommanddde	N/a	
getcommandddeorigin	N/A	
getcurrentdirectory	Transformação direta	Environment.CurrentDirectory
getdatadde	N/A	
getdataddeorigin	N/A	
getenvironment	N/A	
getfileopenname	N/A	
getfilesavename	N/A	
getfocus	N/A	
getfolder	N/A	
getlibrarylist	N/A	
getremote	N/A	
gpf	N/A	
handle	N/A	
hour	Transformação indireta	DateTime.Hour ou Time.Hour
idle	N/A	
imegetcompositiontext	N/A	
imegetmode	N/A	
imesetmode	N/A	

int	Transformação direta	Convert.ToInt16
integer	Transformação direta	Convert.ToInt16
inthigh	Emulação de Função	
intlow	Emulação de Função	
isallarabic	N/A	
isallhebrew	N/A	
isanyarabic	N/A	
isanyhebrew	N/A	
isarabic	N/A	
isarabicandnumbers	N/A	
isdate	Emulação de Função	
ishebrew	N/A	
ishebrewandnumbers	N/A	
isnull	Transformação indireta	== null
isnumber	Emulação de Função	
istime	Emulação de Função	
isvalid	Transformação indireta	!= null
keydown	N/A	
lastpos	Emulação de Função	
left	Transformação indireta	String.Substring
lefta	Transformação indireta	String.Substring
lefttrim	Transformação indireta	String.TrimStart
lefttrimw	Transformação indireta	String.TrimStart
leftw	Transformação indireta	String.Substring
len	Transformação indireta	String.Length
lena	Transformação indireta	String.Length
lenw	Transformação indireta	String.Length
librarycreate	N/A	
librarydelete	N/A	
librarydirectory	N/A	
librarydirectoryex	N/A	
libraryexport	N/A	
libraryimport	N/A	
log	Transformação direta	Math.Log
logten	Transformação direta	Math.Log10
long	Transformação direta	Convert.ToInt32
longlong	Transformação direta	Convert.ToInt64

lower	Transformação indireta	String.ToLower
lowerbound	Transformação indireta	0
match	N/A	
matchw	N/A	
max	Transformação direta	Math.Max
messagebox	N/A	
mid	Transformação indireta	String.Substring
mida	Transformação indireta	String.Substring
mida	Transformação indireta	String.Substring
midw	Transformação indireta	String.Substring
min	Transformação direta	Math.Min
minute	Transformação indireta	DateTime.Minute ou Time.Minute
mod	Transformação indireta	% (operador)
month	Transformação indireta	DateTime.Month ou Date.Month
now	Transformação direta	DateTime.Now
open	N/A	
openchannel	N/A	
opensheet	N/A	
opensheetwithparm	N/A	
openwithparm	N/A	
pi	Transformação direta	Math.PI
pixelstounits	N/A	
populateerror	N/A	
pos	Transformação direta	String.IndexOf
posa	Transformação direta	String.IndexOf
post	N/A	
posw	Transformação direta	String.IndexOf
print	N/A	
print	N/A	
printbitmap	N/A	
printcancel	N/A	
printclose	N/A	
printdatawindow	N/A	
printdefinefont	N/A	
printgetprinter	N/A	
printgetprinters	N/A	
printline	N/A	
printopen	N/A	
printoval	N/A	

printpage	N/A	
printrect	N/A	
printroundrect	N/A	
printscreen	N/A	
printsend	N/A	
printsetfont	N/A	
printsetprinter	N/A	
printsetspacing	N/A	
printsetup	N/A	
printsetupprinter	N/A	
printtext	N/A	
printwidth	N/A	
printx	N/A	
printy	N/A	
profileint	Emulação de Função	
profilestring	Emulação de Função	
rand	Emulação de Função	
randomize	Emulação de Função	
real	Transformação direta	Convert.ToFloat
registrydelete	Emulação de Função	
registryget	Emulação de Função	
registrykeys	Emulação de Função	
registryset	Emulação de Função	
registryvalues	Emulação de Função	
relativedate	Emulação de Função	
relativetime	Emulação de Função	
removedirectory	Emulação de Função	
replace	Emulação de Função	
replacea	Emulação de Função	
replacew	Emulação de Função	
respondremote	N/A	
restart	N/A	
reverse	Emulação de Função	
rgb	N/A	
right	Transformação indireta	String.SubString
righta	Transformação indireta	String.SubString
righttrim	Transformação indireta	String.TrimEnd
righttrimw	Transformação indireta	String.TrimEnd
rightw	Transformação indireta	String.SubString
round	Transformação direta	Math.Round
run	N/A	
run	N/A	

second	Transformação indireta	DateTime.Second ou Time.Second
secondsafter	Emulação de Função	
send	N/A	
setdatadde	N/A	
setlibrarylist	N/A	
setnull	Transformação indireta	= null
setpointer	N/A	
setprofilestring	Emulação de Função	
setremote	N/A	
sharedobjectdirectory	N/A	
sharedobjectget	N/A	
sharedobjectregister	N/A	
sharedobjectunregister	N/A	
showhelp	N/A	
showpopuphelp	N/A	
sign	N/A	
signalerror	N/A	
sin	Transformação direta	Math.Sin
sleep	Transformação direta	Thread.Sleep
space	Transformação indireta	new string(' ', length)
sqrt	Transformação direta	math.Sqrt
starthotlink	N/A	
startserverdde	N/A	
stophotlink	N/A	
stopserverdde	N/A	
string	Transformação indireta	.ToString()
tan	Transformação direta	Math.Tan
time	Transformação direta	Time.Parse
timer	N/A	
toansi	Emulação de Função	
today	Transformação direta	DateTime.Today ou Date.Today
tounicode	Emulação de Função	
toutf8	Emulação de Função	
tracebegin	N/A	
traceclose	N/A	
tracedisableactivity	N/A	
tracedump	N/A	
traceenableactivity	N/A	
traceend	N/A	
traceerror	N/A	
traceopen	N/A	
traceuser	N/A	
trim	Transformação indireta	String.Trim

trimw	Transformação indireta	String.Trim
truncate	Transformação direta	Math.Truncate
unitstopixels	N/A	
upper	Transformação indireta	String.ToUpper
upperbound	Transformação indireta	PBArray.Length
wordcap	N/A	
xmlparsefile	N/A	
xmlparsestring	N/A	
year	Transformação indireta	DateTime.Year ou Date.Year
yield	N/A	
debugbreak	N/A	
garbagecollect	N/A	

Referências

Appel, A. W., & Ginsburg, M. (1997). *Modern compiler implementation in C: basic techniques*. S.l.: Cambridge University Press .

Appeon. (2017). PowerBuilder Roadmap.

Bennett, K. H., Ramage, M., & Munro, M. (1999). Decision model for legacy systems. In *IEE Proceedings-Software* (Vol. 146, pp. 153–159–153–159).

Berner, M. (2016). Appeon Signs Agreement with SAP to Bring Major Innovations to PowerBuilder. Retrieved April 14, 2017, from http://scn.sap.com/community/powerbuilder/blog/2016/07/05/appeon-signs-agreement-with-sap-to-bring-major-innovations-to-powerbuilder

Bianco, R. (2010). The PowerBuilder Phenomenon. Retrieved April 14, 2017, from http://displacedguy.com/the-powerbuilder-phenomenon

Bisbal, J., Lawless, D., Bing Wu, & Grimson, J. (1999). Legacy information systems: issues and directions. *IEEE Software*, *16*(5), 103–111. https://doi.org/10.1109/52.795108

Brandel, M. (2007). The top 10 dead (or dying) computer skills. Retrieved April 14, 2017, from http://www.computerworld.com/article/2541481/data-center/the-top-10-dead--or-dying--computer-skills.html

Brodie, M. L., & Stonebraker, M. (1995). *Legacy Information Systems Migration: Gateways, Interfaces, and the Incremental Approach*. San Francisco: Morgan Kaufmann Publishers Inc.

Brown, M. F., & Armstrong, B. (2003). *PowerBuilder 9: Advanced Client/Server Development*. Sams.

Chisolm, K. C., & Lisonbee, J. C. (1999). The use of computer language compilers in legacy code migration. In *AUTOTESTCON9́9. IEEE Systems Readiness Technology Conference, 1999. IEEE* (pp. 137–145–137–145).

Chomsky, N. (1955). *The logical structure of linguistic theory*. Ms: Plenum.

Cifuentes, C., Simon, D., & Fraboulet, A. (1998). Assembly to high-level language translation. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (pp. 228–237). IEEE Comput. Soc. https://doi.org/10.1109/ICSM.1998.738514

Deitel, H., & Deitel, P. (2016). *C# 6 for Programmers* (6th ed.). Prentice Hall.

Dekkers, C. A. (1999). Pontos de função e medidas: O que é um ponto de função .

El-Ramly, M., Eltayeb, R., & Alla, H. A. (2006). An Experiment in Automatic Conversion of Legacy Java Programs to C#. *Proc. IEEE CSA*, 1037–1045. https://doi.org/10.1109/AICCSA.2006.205215

Engel, J. (1999). *Programming for the Java^{TM} Virtual Machine*. Addison-Wesley Professional.

Fontanette, V., PRADO, A., & OLIVEIRA, A. L. (2004). Uma Abordagem para Migração Gradativa de Aplicações Legadas. Monografia de Qualificação. Departamento de Computação, UFSCar .

Fujiwara, D., Ishiura, N., Sakai, R., Aoki, R., & Ogawara, T. (2016). Reverse engineering from mainframe assembly to C codes in legacy migration. *Proceedings - 2016 5th IIAI International Congress on Advanced Applied Informatics, IIAI-AAI 2016*, 1058–1063. https://doi.org/10.1109/IIAI-AAI.2016.37

Grenning, J. W. (2002). Planning poker or how to avoid analysis paralysis while release planning. *Hawthorn Woods: Renaissance Software Consulting*, *3*.

Hamilton, N. (2008). The A-Z of Programming Languages: C#. Retrieved April 14, 2017, from http://www.computerworld.com.au/article/261958/a-z_programming_languages_c_/?pp=2

Haugen, N. C. (n.d.). An Empirical Study of Using Planning Poker for User Story Estimation. In *AGILE 2006 (AGILE’06)* (pp. 23–34). IEEE. https://doi.org/10.1109/AGILE.2006.16

Hejlsberg, A., Golde, P., Wiltamuth, S., & Torgersen, M. (2010). *The C# Programming Language* (4th ed.). Addison-Wesley Professional.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2001). Introduction to automata theory, languages, and computation. *ACM SIGACT News*, *32*(1), 60–65–60–65.

ISO. (2016). ISO/IEC 9075-2:2016 Information technology -- Database languages -- SQL -- Part 2: Foundation (SQL/Foundation). Retrieved February 27, 2017, from https://www.iso.org/standard/63556.html

K.V.N. Sunitha. (2013). *Compiler Construction*. Pearson India.

Kontogiannis, K., Martin, J., Wong, K., Gregory, R., Müller, H., & Mylopoulos, J. (2010). Code migration through transformations: An experience report. In *CASCON First Decade High Impact Papers* (pp. 201–213–201–213). IBM Corp .

KOSKINEN, J. (2005). Software modernization decision criteria: An empirical study. In *Ninth European Conference on Software Maintenance and Reengineering. IEEE* (pp. 324–331–324–331).

Koskinen, J., Ahonen, J., Lintinen, H., Sivula, H., & Tilus, T. (2004). Estimation of the business value of software modernizations. *Information*.

Ky, J. (2016). *C#: A Beginner's Tutorial* (2nd ed.). Brainy Software.

Landwerth, I. (2016). Introducing .NET Standard. Retrieved April 14, 2017, from https://blogs.msdn.microsoft.com/dotnet/2016/09/26/introducing-net-standard/

LEHMAN, M. M., PERRY, D. E., & RAMIL, J. F. (1998). Implications of evolution metrics on software maintenance. In *Software Maintenance, 1998. Proceedings., International Conference on. IEEE* (pp. 208–217–208–217).

Likert, R. (1932). A technique for the measurement of attitudes. *Archives of Psychology*. https://doi.org/2731047

Martin, J., & Muller, H. A. (2001). Strategies for migration from C to Java. In *Proceedings*

Fifth European Conference on Software Maintenance and Reengineering (pp. 200–209). IEEE Comput. Soc. https://doi.org/10.1109/CSMR.2001.914988

Menezes, P. B. (2011). Linguagens formais e autômatos .

Molokken-Ostvold, K., & Haugen, N. C. (2007). Combining Estimates with Planning Poker--An Empirical Study. In *2007 Australian Software Engineering Conference (ASWEC'07)* (pp. 349–358). IEEE. https://doi.org/10.1109/ASWEC.2007.15

Mossienko, M. (2003). Automated Cobol to Java recycling. In *Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings.* (pp. 40–50). IEEE Comput. Soc. https://doi.org/10.1109/CSMR.2003.1192409

MSDN. (2003). Visual C# Language. Retrieved April 9, 2017, from https://msdn.microsoft.com/en-us/library/aa287558

NEWCOMB, P., & DOBLAR, R. (2001). Automated transformation of legacy systems. *The Journal of*.

Nguyen, T. D., Nguyen, A. T., & Nguyen, T. N. (2016). Mapping API elements for code migration with vector representations. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16* (pp. 756–758). New York, New York, USA: ACM Press. https://doi.org/10.1145/2889160.2892661

Nick Harrison. (2017). *Code Generation with Roslyn*. Apress.

Parr, T. (2009). *Language Implementation Patterns*. Pragmatic Bookshelf.

Parr, T., & Fisher, K. S. (2011). LL(*): The Foundation of the ANTLR Parser Generator. *ACM SIGPLAN Notices, Association for Computing Machinery*, 425–436.

Parsons, J. (2015). Roslyn Overview. Retrieved February 28, 2017, from https://github.com/dotnet/roslyn/wiki/Roslyn Overview

Parthasarathy, M. A. (2007). *Practical software estimation: function point methods for insourced and outsourced projects. Upper Saddle*. River, NJ: Addison-Wesley.

PIRKELBAUER, P. M. (2010). *Programming language evolution and source code rejuvenation* .

Sahin, I., & Zahedi, F. M. (2001). Policy analysis for warranty, maintenance, and upgrade of software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, *13*(6), 469–493–469–493.

Sybase. (2011). *DataWindow® Reference - PowerBuilder® Classic 12.5*. Sybase. Retrieved from http://infocenter.sybase.com/help/topic/com.sybase.infocenter.dc37783.1250/pdf/dwref.pdf

Sybase. (2012a). *PowerBuilder® 12.5 Users Guide*. Sybase. Retrieved from http://infocenter.sybase.com/help/topic/com.sybase.infocenter.dc00844.1250/pdf/pbug.pdf

Sybase. (2012b). *PowerScript® Reference - PowerBuilder® Classic 12.5*. Sybase. Retrieved from http://infocenter.sybase.com/help/topic/com.sybase.infocenter.dc37781.1250/pdf/psref.pdf

Terekhov, A. A. (2001). Automating language conversion: a case study (an extended abstract). In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001* (pp. 654–658). IEEE Comput. Soc. https://doi.org/10.1109/ICSM.2001.972782

TIOBE. (2016). TIOBE Index | TIOBE - The Software Quality Company - Disponível em: <http://www.tiobe.com/tiobe-index>. Acesso em 3 de dez.

Ullman, J. D., Aho, A. V, & Sethi, R. (1995). Compiladores–Princıpios Técnicas e Ferramentas .

Ulrich, W. (2004). A status on OMG architecture-driven modernization task force. In *Proceedings EDOC Workshop on Model-Driven Evolution of Legacy Systems (MELS).*

IEEE Computer Society Digital Library .

Verhoef, C., & Terekhov, A. A. (2000). The realities of language conversions. *IEEE Software*, *17*(6), 111–124. https://doi.org/10.1109/52.895180

Visaggio, G. (2000). Value-based decision model for renewal processes in software maintenance. *Annals of Software Engineering*, *9*(1–2), 215–233–215–233.

Warren, I., & Ransom, J. (2002). Renaissance: a method to support software system evolution. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International* (pp. 415–420–415–420).