Universidade Federal de São Carlos
Departamento de Computação

Maria Janaina da Silva Ferreira Giocondo

Desenvolvimento de um sistema configurável para tratamento de erros em compiladores

Projeto de pesquisa para qualificação de mestrado apresentada ao Departamento de Computação como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Área de concentração: Computação.

Orientador: Prof. Dr. José de Oliveira Guimarães

Sorocaba
2013

1 Resumo

Durante o processo de desenvolvimento de software, as mensagens exibidas pelos compiladores são de extrema importância para entendimento e tratamento dos erros encontrados durante o processo de compilação.

A forma como a mensagem é exibida afeta de maneira significativa o desempenho e interpretação do programador. Em particular dos programadores iniciantes e estudantes de computação, já que os programadores mais experientes possuem um pré conhecimento sobre a situação pois já vivenciaram o erro anteriormente.

Muito se tem feito hoje em dia para favorecer o desenvolvimento de softwares, através da criação de interfaces amigáveis e visuais que facilitam o processo de desenvolvimento, edição, compilação e exibição de programas de computação. Entretanto, pouco se tem trabalhado para melhorar as mensagens de erros dos compiladores, usualmente exibidas com termos difíceis, prejudicando a compreensão, resolução e prevenção do erro ocorrido.

Dessa forma, torna-se necessário uma solução que permita a melhor interpretação do erro e logo um maior aumento na produtividade e qualidade de desenvolvimento do produto de software. Esse projeto propõe o desenvolvimento de uma ferramenta a ser acoplada ao compilador da linguagem Cyan em que as mensagens de erro serão amigáveis e acompanhadas de exemplos do próprio código do usuário que causou o erro (mostrando o erro e o código correto, se possível). O programador poderá armazenar as suas próprias explicações e comentários para cada erro. Existirão várias linguagens específicas de domínio que permitirão uma configuração completa da ferramenta.

2 Introdução

O processo de desenvolvimento de um software é minucioso e demanda muito tempo a fim de se obter um produto final de qualidade [49]. Neste contexto, muito se tem feito para melhorar esse processo através da implementação de novas funcionalidades e linguagens de programação mais adequadas. Por outro lado pouco se tem feito para melhorar a exibição dos erros e facilitar a compreensão dos programadores sobre as mensagens apresentadas.

Os primeiros trabalhos sobre o tema são de 1983 [7]. A partir desta data poucas pesquisas foram realizadas a fim de resolver o problema. As pesquisas atuais focam na disciplina de Interação Humano Computador (IHC) [61] [62], apresentando um modelo para a exibição das mensagens. Elas partem do pressuposto que a forma como as mensagens são exibidas afetam diretamente o aprendizado e o processo de desenvolvimento do software.

A tarefa de desenvolvimento em si exige muita concentração e raciocínio e o esforço para interpretar o que o compilador quer dizer torna o aprendizado mais lento e cansativo. Essas pesquisas sugerem, mas não oferecem, uma solução prática para o problema.

Os compiladores atuais também não possuem ajuda adicional além da mensagem exibida, o que induz os programadores a procurarem formas alternativas de auxílio, em fóruns ou grupos em redes sociais. Essa busca faz com que sua dúvida não seja solucionada rapidamente, levando o programador a usar uma solução muitas vezes não adequada. Até mesmo os compiladores modernos focam na premissa que o programador tem que se adaptar ao compilador.

Um dos objetivos dessa pesquisa é o desenvolvimento de uma ferramenta que permita que o compilador se adapte ao usuário. Esta ferramenta será configurada pelo usuário através de linguagens específicas de domínio (LED).

3 Justificativa

As mensagens de erro dos compiladores não possuem um padrão de exibição e são diferentes em cada compilador, além de possuírem termos de difícil compreensão. O programador tem que recordar das ações que ele tomou quando o compilador emitiu uma mensagem igual à atual no passado.

A interpretação não adequada da mensagem pode induzir o programador a buscar formas alternativas para corrigir o problema, diminuindo assim sua produtividade e a qualidade do produto de software que está sendo desenvolvido.

Traver [62] sugere algumas soluções para preenchimento dessas lacunas, como a sinalização do local exato do erro, a exibição de mensagens adequadas com o nível de conhecimento do programador, a possibilidade de consulta aos erros anteriormente ocorridos e o obtenção da solução se não imediatamente, mas em menor tempo possível.

Este trabalho prevê o desenvolvimento de uma ferramenta para atender as soluções propostas por Traver. Assim o desenvolvimento de software pode ser mais rápido e com mais qualidade porque a interação entre o programador e o computador seria melhorada.

4 Objetivos

O compilador Cyan será acoplado ao ambiente de desenvolvimento Eclipse [59] de tal forma que se possa editar código, compilar e ver as mensagens no IDE. Este projeto consiste em construir parte de uma ferramenta chamada EMS (Error Message System) que será um plugin para o Eclipse. Esta ferramenta permitirá que, quando o compilador sinalizar um erro, este seja explicado detalhadamente ao programador Cyan, apresentando as possíveis causas do erro, códigos anteriores que apresentavam o mesmo erro, sugestões de correção e ligações visuais entre os trechos de código envolvidos no erro. EMS é um grande projeto do qual esta dissertação é um sub-projeto.

Esta seção apresenta todo o projeto EMS para que o leitor tenha uma visão geral do resultado final. O texto que se segue é a descrição da arquitetura da ferramenta EMS com a visão do usuário final sobre ela. O trabalho da discente será especificar algumas linguagens específicas de domínio descritas abaixo, implementá-las e acoplar opções ao Eclipse para que o subconjunto da ferramenta EMS de que trata o mestrado seja acessível graficamente. Exatamente o que será feito é especificado ao fim desta seção. Espera-se que este projeto seja utilizado para mais de uma dissertação de mestrado.

O compilador utilizado neste projeto é o da linguagem Cyan, implementado em Java, a mesma linguagem que será utilizada para fazer o plugin para o Eclipse. Este compilador não está pronto, mas há um subconjunto dele que já produz mensagens de erro suficientes para este projeto. É importante notar que não há nada específico na arquitetura geral de EMS em relação a Cyan ou Java. Este projeto poderia não só ser adaptado a outras linguagens como também a outros sistemas que sinalizam erros que podem ser difíceis de entender (não todos os sistemas com esta característica, mas alguns deles). É apresentado a seguir uma visão geral do EMS seguido de subseções no qual esta ferramenta é detalhada.

Será utilizado o termo "erro de compilação" ou "tipo de erro" como referência geral a um erro como "variável não declarada". Será utilizado "instância do erro" para uma ocorrência específica sinalizada pelo compilador. Esta instância está associada a uma linha de código (em geral) e a um arquivo específico de Cyan.

A Figura 1 mostra os relacionamentos entre os elementos da EMS. Uma seta da caixa A para B com um 1 na origem e $\star$ no destino mostra a associação de um exemplar de A com zero ou mais exemplares de B. O elemento central deste trabalho é "Erro de Compilação", um erro de compilação (veja o quadro cinza). Para cada tipo de erro de compilação como "método duplicado", "variável não declarada", "ponto e vírgula ausente" ou "método declarado com tipo de retorno incorreto" estão associados:

(a) "Parâmetros específicos para este erro" (veja caixa com este conteúdo), que são informações específicas sobre este erro de compilação e o ambiente onde o erro aconteceu. Por exemplo, o erro "variável não declarada" está associado à informação do nome da variável, o método onde ela está sendo utilizada e o protótipo [1] no qual o método está. Estas informações são os parâmetros específicos para este erro. Cada tipo de erro tem os seus parâmetros específicos que podem ser os nomes do método e protótipo onde o erro foi sinalizado, a instrução do erro, o nome do super-protótipo etc;

(b) "Texto com a explicação do erro". Quando ocorre um erro de compilação, será apresentado ao programador um texto adaptado àquele trecho de código que causou o erro. Como exemplo, ao

[1] Em Cyan todas as classes são consideradas protótipos

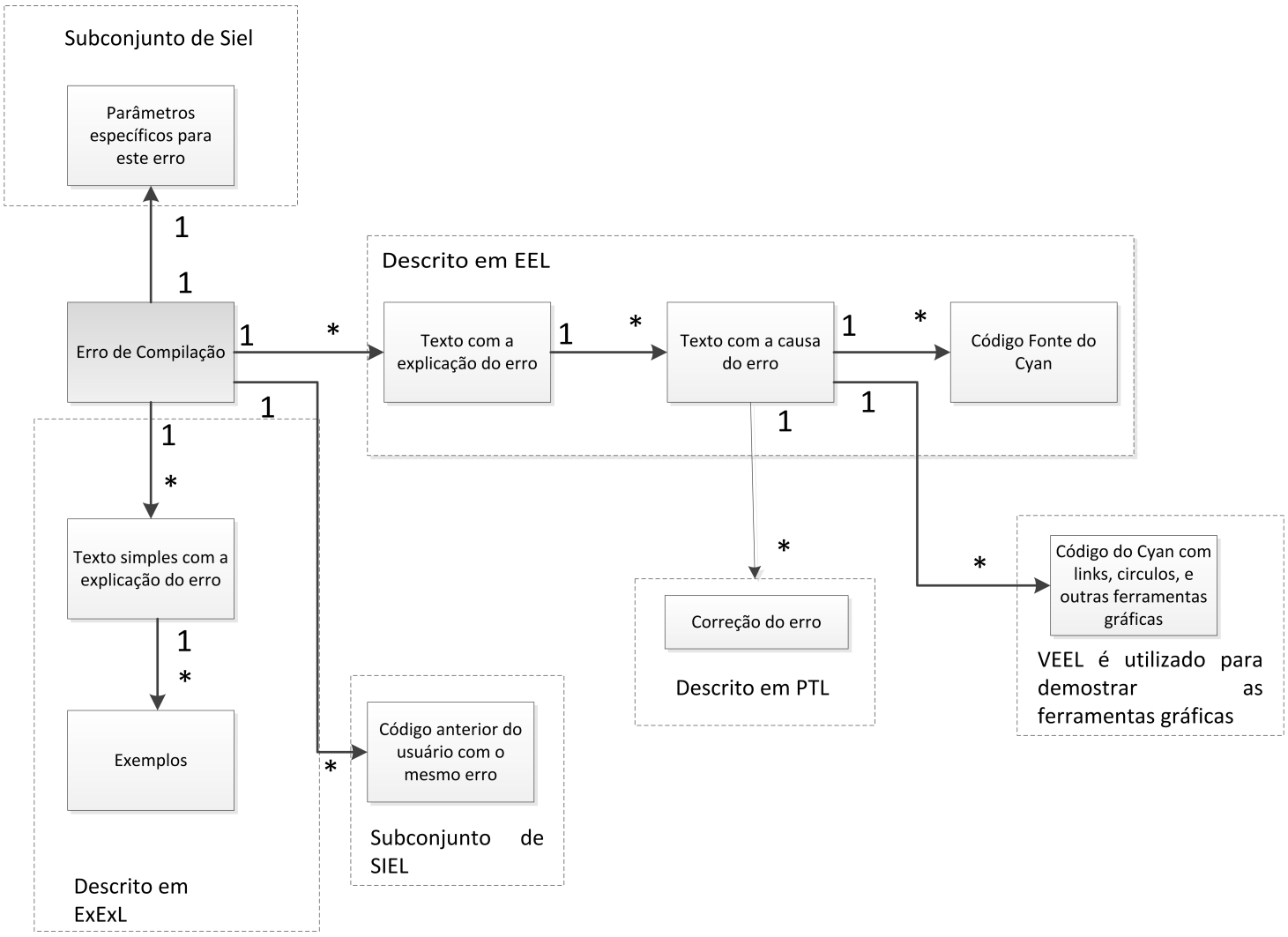

Figura 1: Esquema do projeto

invés do compilador sinalizar "variável não declarada" ele apresentará a mensagem "variável 'a' no método 'scan' do protótipo 'List' não foi declarada". Esta parte do EMS poderá mostrar mais do que uma linha de erro adaptada, com os nomes utilizados no código do usuário: poderá ser mostrado todo o contexto do erro com as suas diversas causas e possíveis correções (novamente, com os nomes empregados nesta instância específica deste erro);

(c) "Texto simples com a explicação do erro", um ou mais textos com a explicação do erro. Quando ocorre um erro de compilação, o programador poderá ver estes textos que explicarão o erro de uma maneira geral, sem referências a identificadores específicos que causaram esta instância específica do erro. Como exemplo, se o erro de compilação foi "variável não declarada", o texto correspondente a "Texto simples com a explicação do erro" não conterá o nome da variável ou o do método onde o erro ocorreu. A explicação será geral e não adaptada à instância específica deste erro no código Cyan do usuário. Este tipo de explicação é mais difícil de entender, pela falta de contexto, do que a de "Texto com a explicação do erro". Mas é mais fácil de fazer e pode ser semi-automatizada;

(d) "Código anterior do usuário com o mesmo erro". Para cada erro de compilação, pode haver zero ou mais links para códigos compilados anteriormente com o mesmo tipo de erro. Quando

um erro de compilação acontece, o usuário poderá escolher armazenar este erro na sua própria base de dados de erros, juntamente com suas próprias observações a respeito dele.

As explicações detalhadas dos elementos da Figura 1 são dadas nas subseções seguintes.

4.1 Parâmetros específicos para este erro

O compilador Cyan sinalizará um erro de compilação chamando o método `signalCompilerError` da classe `Compiler` (o compilador é feito em Java). Este método toma um número variável de *strings* como parâmetro:

```
void signalCompilerError(String ... info)
```

Estes parâmetros fornecem, além do tipo do erro (como "variável não declarada"), informações do erro (o nome da variável não declarada é "`a`") e do contexto onde ele ocorreu (o nome do método e protótipo onde foi sinalizado o erro como "`search`" e "`PersonList`").

A concatenação das strings do vetor `info`[2] compõe o código de uma Linguagem Específica de Domínio (LED). Cada tipo de erro estará associado a uma LED diferente que é basicamente um subconjunto de uma LED mais geral chamada de SIEL (*Specific information error language*). Código em SIEL consiste de pares **nome-valor** cada um deles contendo uma descrição de algum elemento do programa do usuário (maiores explicações a seguir).

Os parâmetros extraídos do código da LED contido na string `info` serão utilizados na apresentação das explicações para este erro, nas suas possíveis causas etc. Por este motivo cada tipo de erro precisa de uma LED diferente. Como exemplo, o erro "variável não declarada" terá como informação o tipo do erro "`Variable was not declared`", o método onde a variável está sendo utilizada[3], o protótipo onde está este método, as variáveis locais visíveis no ponto onde está esta atribuição e as variáveis de instância do protótipo.

Esta LED é basicamente uma lista de pares atributos/valores. As LED's para outros erros são similares a estas podendo requerer atributos diferentes. Por exemplo, para o erro "ponto e vírgula ausente" não é necessário o atributo `instanceVariables` que descreve as variáveis de instância do protótipo sendo compilado. Mas certamente seria necessário o atributo `lastStatement` que é uma string com o texto da última instrução.

4.2 Texto simples com a explicação do erro

Cada erro de compilação pode estar associado a um texto que o explica. E cada um destes textos pode estar associado a zero ou mais exemplos. Um único texto com os seus exemplos é um código em uma LED chamada ExExL. A descrição formal desta linguagem fará parte da dissertação de mestrado.

O código em ExExL com a explicação e os exemplos deverão ser gerados por uma ferramenta gráfica ExExLWrite incorporada ao Eclipse. O usuário fornecerá a explicação, língua, exemplos de código etc e ExExLWrite produzirá o código em ExExL. Haverá uma outra ferramenta, ExExL-Read que lê código em ExExL e o incorpora ao compilador. E ExExLManagment que gerencia os códigos em ExExL associados a cada erro. Em particular, esta ferramenta permitirá que vários códigos em ExExL sejam incorporados a diversos tipos de erro de compilação de uma só vez.

[2] `info` é tratado como um vetor de `String` em Java.

[3] Assuma que é em uma atribuição como "`a = expr`" e a variável não declarada seria "`a`".

4.3 Código anterior do usuário com o mesmo erro

Para cada erro de compilação, pode haver zero ou mais ligações para códigos com o mesmo tipo de erro. Quando um erro de compilação acontece, como "herança ilegal", o usuário poderá escolher armazenar este erro na sua própria base de dados de erros, juntamente com suas próprias observações a respeito dele. O que deve ser guardado é especificado por um subconjunto de SIEL. O mesmo subconjunto utilizado em "Parâmetros específicos para este erro".

A diferença entre as funcionalidades "Código anterior do usuário com o mesmo erro" e "Texto simples com a explicação do erro" é que na primeira o código é extraído automaticamente do programa do usuário, bastando para isto apenas um clique. No segundo o programador escolhe quais trechos ele disponibilizará, podendo inclusive apresentar a correção. No primeiro, a versão final, corrigida, não é automaticamente incluída porque o programador, ao corrigir o erro, poderá ter alterado completamente o programa, tornando impossível para a ferramenta localizar as modificações que tornaram o código Cyan correto.

4.4 Texto com a explicação do erro

Para cada erro de compilação serão associados um ou mais textos com a explicação do erro, sendo que um deles é o padrão. Ao acontecer o erro, ao programador será apresentada a explicação padrão com a opção de ver as explicações não padrões e inclusive tornar uma outra como a padrão para aquele tipo de erro. Estas explicações usarão os identificadores presentes naquela instância do erro ("variável 'a' no método 'search' do protótipo 'PersonList' não foi declarada" ao invés de "variável não declarada"). Para cada explicação haverá zero ou mais textos com a causa do erro. Cada erro de compilação pode ter várias causas diferentes. Por exemplo, uma atribuição "`a = b`" pode fazer o compilador sinalizar o erro "variável não declarada". As razões podem ser "a variável não foi declarada localmente", "a variável não foi declarada como variável de instância", "a variável foi digitada incorretamente". Apesar destas causas serem apenas variações de uma mesma, utilizaremos este exemplo pela sua simplicidade de entendimento.

Em geral, a maioria dos erros terá apenas uma única causa. Mas alguns, como erros sintáticos, frequentemente podem ser causados por diversos fatores.

Para cada **causa do erro**, estão associados:

(i) zero ou mais trechos de código em Cyan (veja retângulo "Código Fonte Cyan" na Figura). Estes trechos mostram as partes do código que causaram ou poderiam ter causado o erro. No exemplo "variável não declarada", a causa "a variável não foi declarada localmente" estaria associada a dois trechos de código, que são a atribuição "`a = b`" que originou o erro e a lista de declarações de variáveis locais;

(ii) zero ou mais correções para o erro (veja retângulo "Correção do erro"). Se disponível, o programador poderá aceitar uma das sugestões dadas pelo compilador para corrigir o erro. No erro "variável não declarada", causa "a variável não foi declarada localmente", o compilador poderá apresentar as sugestões de declarar a variável como local ou parâmetro com um tipo que ele sugere (se não for possível deduzir um tipo apropriado). Para a causa "a variável foi digitada incorretamente" o compilador poderá sugerir mudar o nome da variável para o nome de uma variável local ou de instância já declarada. O compilador pode até indicar alguma variável cujo nome seja próximo (por algum critério) ao da variável não declarada (o erro teria sido causado por um erro de digitação, "pesronList" ao invés de "personList");

(iii) zero ou mais trechos de código em Cyan nos quais as partes importantes são realçados com círculos, ligações, setas etc. Estes elementos gráficos poderão não só indicar as causas do erro como também mudanças no código para corrigi-lo. No erro "variável não declarada", o compilador poderá apresentar um trecho de código com a declaração de todas as variáveis de instância, outro com todas as variáveis locais e parâmetros. Os nomes das variáveis declaradas nestes lugares poderiam estar realçados com uma cor diferente ou envolvidos por um círculo. Variáveis cujos nomes são próximos do nome da variável que não foi declarada poderiam ser apresentados com uma cor diferente.

Os retângulos "*Texto com a explicação do erro*", "*Texto com a causa do erro*" e "*Código fonte do Cyan*" estão dentro de um retângulo com bordas pontilhadas indicando que as informações especificadas destes retângulos serão descritas através de uma linguagem chamada de EEL (Linguagem de explicação do erro). Para cada erro de compilação, caixa "Erro de Compilação", haverá um ou mais códigos em EEL, cada um deles contendo: (a) um texto com a explicação do erro ("Texto com a explicação do erro"); (b) zero ou mais textos especificando quais as possíveis causas do erro ("Texto com a causa do erro") e (c) para cada causa, zero ou mais códigos em Cyan ("Código fonte do Cyan"). O código em EEL utiliza informações passadas pelo compilador ao método `signalCompilerError` que são específicas a cada erro. Então o código em EEL é parametrizado — para cada tipo de erro os parâmetros são diferentes.

No restante do código em EEL os parâmetros poderiam ser utilizados. O texto com a explicação do erro, por exemplo, poderia se referir ao nome da variável que não foi declarada, que é o parâmetro `variableName`. Este texto poderia ser "*variável '#variableName' no método '#methodName' do protótipo '#prototypeName' não foi declarada*". A ferramenta substituiria *#variableName* pelo valor do parâmetro `variableName` antes de apresentar a mensagem. Idem para os outros parâmetros, o que resultaria em uma mensagem "variável 'a' no método 'search' do protótipo 'PersonList' não foi declarada".

Quando o compilador detectar que a variável "`a`" em "`a = b`" não foi declarada, ele chama o método `signalCompilerError` passando um código de um subconjunto de SIEL contendo informações sobre o nome da variável não declarada, o método e o protótipo onde o erro foi detectado etc. Recordando, para cada erro há uma LED com pares atributos/valores importantes para aquele erro. E esta LED é um subconjunto de SIEL (veja caixa "Parâmetros específicos para este erro").

O método `signalCompilerError` procurará o código em EEL associado a este erro — esta busca será feita em uma tabela que contém pares Erro/Código EEL, sendo "Erro" a string que é o valor do atributo `error` do código do subconjunto de SIEL passado como parâmetro na chamada a `signalCompilerError`. Na chamada

```
signalCompilerError( "error = 'Variable was not declared'", ... );
```

este método faria a seguinte busca:

```
eelCode = hashTable.get("Variable was not declared");
```

Por motivos didáticos, substituimos o valor associado ao atributo `error` por uma string literal neste exemplo.

Após recuperar o código em EEL, o método `signalCompilerError` o executará passando os parâmetros que recebeu do compilador. Apesar do código em EEL ser largamente descritivo, ele é executável, pois ordena ao compilador mostrar os textos com explicação, as causas do erro, os códigos em Cyan etc. A ferramenta mostrará estas infomações utilizando janelas, listas, textos etc. O modo como ele fará isto não será precisamente especificado neste projeto pois a melhor forma só será encontrada durante a construção da interface. O código em EEL será transformado

em uma Árvore de Sintaxe Abstrata e, para apresentar os elementos gráficos, um método da raiz será chamado. Este método poderá apresentar algum elemento gráfico e então invocar métodos de nós filhos. O processo se repete com os filhos.

A busca pelo código EEL na tabela retornará o código padrão, que sempre existirá. Mas podem existir outros alternativos. Idealmente estes códigos poderão ser adquiridos de outros usuários e acoplados ao compilador.

O código em EEL é um *template*, esqueleto com nomes que são substituídos pela instância específica do erro. Desta forma, quando o compilador sinalizar um erro no código em Cyan, ele apresentará uma explicação e código com os nomes utilizados no código Cyan (nome da variável, do método etc). Este código apresentado ao programador conterá todos os elementos essenciais deste tipo de erro e utilizará os identificadores desta instância específica do erro. O usuário do compilador Cyan terá a impressão que o seu próprio código foi selecionado e apresentado na explicação do erro e nos exemplos desta explicação.

Como exemplo, suponha que o erro "variável não declarada" tenha sido sinalizado pelo compilador, com esta mesma mensagem de erro, que pode ser suficiente para a identificação do erro. Se não for suficiente, o usuário do IDE Eclipse terá a opção de ver uma explicação mais detalhada do erro (código EEL), textos simples com explicações (código ExExL) ou códigos anteriores com o mesmo erro (código CEL). Na primeira opção, a explicação apresentada ao usuário será

```
variável 'a' no método 'search: String' do protótipo
PersonList não foi declarada. Em Cyan, todas as variáveis
...
```

Ou seja, a explicação será adaptada ao contexto desta instância de erro (está sendo utilizado o exemplo de erro dado anteriormente). Se o usuário do IDE escolher ver a causa "a variável 'a' não foi declarada localmente", serão apresentados os códigos seguintes, em duas janelas diferentes:

```
// código 1
         fun search: (:s String) [
            // nenhuma declaração para 'a'
            ...
            a = b;
        ]

// código 2
        // T é um tipo
        fun search: (:s String) [
            :a T;
            ...
            a = b;
        ]
```

Há duas caixas na Figura 1 ainda não descritas:

(a) "Correção do erro". Trata-se de código escrito em uma linguagem chamada PTL (Programa para tranformação da linguagem) para corrigir o código Cyan, eliminando o erro de compilação. Para cada causa de erro estaria associado um código em PTL com a correção. Esta

linguagem teria comandos para modificar a Árvore de Sintaxe Abstrata construída pelo compilador ou mesmo o texto do código Cyan. Esta parte do projeto é muito complexa e não apresentaremos sequer os seus requisitos mínimos;

(b) "Código do Cyan com links, circulos, e outras ferramentas gráficas". A cada causa de erro estará associado um código escrito em uma linguagem chamada VEEL (Linguagem visual para explicação do erro). Este código será responsável por apresentar graficamente o erro, ressaltando partes importantes no código Cyan (com cores diferentes), colocando círculos em identificadores importantes para o erro e ligando visualmente elementos relacionados no código Cyan. Apesar de uma descrição básica de VEEL não ser difícil, esta linguagem não será utilizada neste projeto de mestrado.

Apenas parte do sistema EMS (Error Message System) será desenvolvido na dissertação de mestrado. Esta parte consiste em:

(a) fazer um plugin do Eclipse para que este chame o compilador Cyan e capture as suas mensagens de erro;

(b) seleção de dez tipos de erro do compilador Cyan;

(c) definição da linguagem SIEL e seu compilador;

(d) definição de um subconjunto de SIEL para cada tipo de erro;

(e) definição da linguagem ExExL, seu compilador e das ferramentas ExExLWrite, ExExLRead e ExExLManagment;

(f) definição da linguagem EEL e seu compilador, que construirá uma Árvore de Sintaxe Abstrata (ASA) do código EEL. Esta ASA será percorrida para mostrar o texto com a explicação de cada erro (substituindo os parâmetros pelos valores reais presentes na instância do erro), as causas do erro e os exemplos de código associados a cada causa.

Para trabalhos futuros o projeto prevê uma interface adaptativa, partindo da premissa de que existem vários níveis de usuário e com conhecimentos distintos, as mensagens se adaptariam ao usuário específico e caso o programador tentasse várias vezes insistir no erro sem alterações, as mensagens mudariam à fim de criar um ambiente melhor de interação entre o usuário e o compilador;

Poderão existir repositórios de código ExExL na www. Em particular, no projeto EMS está prevista uma rede social chamada CyPeople incorporada ao Eclipse no qual os programadores Cyan poderão interagir entre si. Através desta rede social os usuários poderão trocar códigos, explicações sobre os erros etc. Em particular, eles poderão trocar alguns dos códigos escritos em LED associados a este projeto, como ExExL, EEL, PTL e VEEL.

CyPeople deverá ter pelo menos as seguintes características:

- será integrada ao Eclipse;
- cada programador poderá escolher "compartilhar" os seus próprios códigos com erro na rede social. Assim, quando ocorrer um erro, será mostrado uma opção "mostre códigos de outros usuários com este mesmo erro". Os códigos dos participantes serão especificados utilizando-se uma linguagem SIEL — veja retângulo "Código anterior do usuário com o mesmo erro";

O objetivo inicial de CyPeople será trocar experiências sobre erros de compilação, mas esta rede social também servirá para troca de experiência sobre programação.

5 Plano de Trabalho

O plano de trabalho consiste de duas partes: estudo e construção de parte do EMS. O estudo envolve fundamentos de IHC, engenharia de software, linguagens de programação e compiladores. A construção da ferramenta envolve o projeto de várias linguagens específicas de domínio e o desenvolvimento de vários pequenos compiladores para estas LED's. Este último passo envolve o projeto da interface gráfica da parte do EMS desenvolvido neste trabalho.

É interessante observar que há um artigo de Traver [62] que será a base de toda a dissertação. Este artigo descreve todos os problemas com os compiladores atuais em relação às mensagens de erro e apresenta uma extensa bibliografia a respeito. Contudo, nem Traver nem as referências citadas propõem soluções adequadas para os problemas.

6 Cronograma das Atividades

O cronograma apresentado abaixo considera que o projeto de mestrado se iniciou em Agosto de 2013 e terá a duração de 12 meses. O trabalho consiste de sete itens, sendo que o terceiro deles contempla o desenvolvimento de um plugin para o Eclipse para que este chame o compilador Cyan e capture as suas mensagens de erro. Contudo, um sistema enorme como o Eclipse é difícil de compreender e o plugin pode não funcionar adequadamente. Se isto acontecer, será feito um IDE simples que permite chamar o compilador Cyan e mostrar as mensagens de erro. O subconjunto de EMS desta dissertação será acoplada então a este IDE.

Atividades/Meses	01	02	03	04	05	06	07	08	09	10	11	12
1	x	x										
2	x	x	x									
3			x	x								
4					x	x						
5								x				
6						x	x	x	x	x	x	
7												x

(1) Revisão bibliográfica. Neste projeto é apresentado todas as obras que serão citadas na dissertação;

(2) Definição das linguagens e desenvolvimento da ferramenta;

(3) Desenvolvimento de plugin do Eclipse para que este chame o compilador Cyan e capture as suas mensagens de erro;

(4) Testes e coleta de dados;

(5) Análise dos resultados;

(6) Redação da dissertação;

(7) Defesa.

7 Forma de Análise dos Resultados

Para análise e confirmação dos resultados, a ferramenta desenvolvida neste projeto será testada por pequenos grupos de programadores cujas sugestões poderão causar modificações não só neste software como em todo o projeto, incluindo as linguagens específicas de domínio.

Referências

[1] A. Alexandrescu. Better template error messages. *C/C++ Users J.*, 17(3):37–47, Mar. 1999.

[2] J. R. Anderson and R. Jeffries. Novice lisp errors: undetected losses of information from working memory. *Hum.-Comput. Interact.*, 1(2):107–131, June 1985.

[3] K. Beck. *Extreme Programming Explained: Embrace Change*. The XP Series. Addison-Wesley, 2000.

[4] B. Beizer. Software is different. *Ann. Softw. Eng.*, 10(1-4):293–310, Jan. 2000.

[5] J. Bonar and E. Soloway. Preprogramming knowledge: a major source of misconceptions in novice programmers. *Hum.-Comput. Interact.*, 1(2):133–161, June 1985.

[6] R. Brooks. Towards a theory of the cognitive processes in computer programming. *Int. J. Hum.-Comput. Stud.*, 51(2):197–211, Aug. 1999.

[7] P. J. Brown. Error messages: the neglected area of the man/machine interface. *Commun. ACM*, 26(4):246–249, Apr. 1983.

[8] C. Burrell and M. Melchert. Augmenting compiler error reporting in the karel++ microworld. *Proceedings of the 20th Annual Conference of the National Advisory Committee on Computing Qualifications*, pages 41–46, July 2007.

[9] J. J. Canas, M. T. Bajo, and P. Gonzalvo. Mental models and computer programming. *Int. J. Hum.-Comput. Stud.*, 40(5):795–811, May 1994.

[10] D. Carrington, B. McEniery, and D. Johnston. Pspsm in the large class. In *Software Engineering Education and Training, 2001. Proceedings. 14th Conference on*, pages 81 –88, 2001.

[11] H. L. Colin Laplace, Mike Berg. Bloodshed software - mingw c++ compiler.

[12] S. Cooper, W. Dann, and R. Pausch. Teaching objects-first in introductory computer science. *SIGCSE Bull.*, 35(1):191–195, Jan. 2003.

[13] N. Coull and U. of St Andrews. Department of Computer Science. Snoopie: Development of a learning support tool for novice programmers within a conceptual framework. 2008.

[14] W. Dann and S. Cooper. Alice: an educational software that teaches students computer programming in a 3d environment.

[15] J. de Oliveira Guimarães. Learning compiler construction by examples. *SIGCSE Bull.*, 39(4):70–74, Dec. 2007.

[16] J. de Oliveira Guimarães. The cyan language. *CoRR*, abs/1306.1870, 2013.

[17] A. Dix. *Human-Computer Interaction*. Pearson/Prentice-Hall, 2004.

[18] A. Ebrahimi. Novice programmer errors: language constructs and plan composition. *Int. J. Hum.-Comput. Stud.*, 41(4):457–480, Oct. 1994.

[19] N. El Boustani and J. Hage. Improving type error messages for generic java. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 131–140, New York, NY, USA, 2009. ACM.

[20] T. Flowers, C. Carver, and J. Jackson. Empowering students and building confidence in novice programmers through gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T3H/10 – T3H/13 Vol. 1, oct. 2004.

[21] K. E. Gray and M. Flatt. Professorj: a gradual introduction to java through language levels. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 170–177, New York, NY, USA, 2003. ACM.

[22] D. Grune. *Modern compiler design*. Worldwide series in computer science. John Wiley, 2000.

[23] D. Grune and C. Jacobs. *Parsing techniques: a practical guide*. Ellis Horwood series in computers and their applications. Ellis Horwood, 1990.

[24] D. Grune and C. Jacobs. *Parsing techniques: a practical guide*. Monographs in computer science. Springer Science+Business Media, LLC, 2008.

[25] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1019–1028, New York, NY, USA, 2010. ACM.

[26] B. Heeren. Improving type-error messages in functional languages. Technical report, 2002.

[27] J. Horning. What the compiler should tell the user. In F. Brauer, J. Eickel, F. Remer, M. Griffiths, U. Hill, J. Horning, C. Koster, W. McKeeman, P. Poole, and W. Waite, editors, *Compiler Construction*, volume 21 of *Lecture Notes in Computer Science*, pages 525–548. Springer Berlin Heidelberg, 1974.

[28] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting java programming errors for introductory computer science students. *SIGCSE Bull.*, 35(1):153–156, Jan. 2003.

[29] W. Humphrey. *Introduction To the Personal Software Process.* SEI Series in Software Engineering. Addison-Wesley Pub., 1997.

[30] J. Jackson, M. Cobb, and C. Carver. Identifying top java errors for novice programmers. In *Frontiers in Education, 2005. FIE '05. Proceedings 35th Annual Conference*, page T4C, oct. 2005.

[31] M. C. Jadud. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the second international workshop on Computing education research*, ICER '06, pages 73–84, New York, NY, USA, 2006. ACM.

[32] C. L. Jeffery. Generating lr syntax error messages from examples. *ACM Trans. Program. Lang. Syst.*, 25(5):631–640, Sept. 2003.

[33] M. Kolling. Bluej the interactive java environment. http://www.bluej.org.

[34] M. Kolling, B. Quig, A. Patterson, and J. Rosenberg. The bluej system and its pedagogy. *Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology*, 13(4):249–268, December 2003.

[35] M. Kolling and J. Rosenberg. Blue a language for teaching object-oriented programming. *SIGCSE Bull.*, 28(1):190–194, Mar. 1996.

[36] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. *SIGPLAN Not.*, 42(6):425–434, June 2007.

[37] C. R. Litecky and G. B. Davis. A study of errors, error-proneness, and error diagnosis in cobol. *Commun. ACM*, 19(1):33–38, Jan. 1976.

[38] G. A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 101(2):343–352, May 1955.

[39] P. G. Moulton and M. E. Muller. Ditran a compiler emphasizing diagnostics. *Commun. ACM*, 10(1):45–52, Jan. 1967.

[40] C. Murphy, G. Kaiser, K. Loveland, and S. Hasan. Retina: helping students and instructors based on observed programming activities. *SIGCSE Bull.*, 41(1):178–182, Mar. 2009.

[41] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander. Debugging: the good, the bad, and the quirky – a qualitative analysis of novices' strategies. *SIGCSE Bull.*, 40(1):163–167, Mar. 2008.

[42] R. Navarro-Prieto and J. J. Cañas. Are visual programming languages better? the role of imagery in program comprehension. *Int. J. Hum.-Comput. Stud.*, 54(6):799–829, June 2001.

[43] J. Nielsen and L. R. Mack. *Usability inspection methods*. John Wiley and Sons, New York, NY, USA, 1994.

[44] M.-H. Nienaltowski, M. Pedroni, and B. Meyer. Compiler error messages: what can help novices? *SIGCSE Bull.*, 40(1):168–172, Mar. 2008.

[45] D. Norman. *The Design of Everyday Things*. The Mit Press, 1998.

[46] J. F. Pane, B. A. Myers, and L. B. Miller. Using hci techniques to design a more usable programming system. In *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, HCC '02, pages 198–, Washington, DC, USA, 2002. IEEE Computer Society.

[47] J. F. Pane, B. A. Myers, and C. A. Ratanamahatana. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Hum.-Comput. Stud.*, 54(2):237–264, Feb. 2001.

[48] R. W. Picard. *Affective computing*. MIT Press, Cambridge, MA, USA, 1997.

[49] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th edition, 2001.

[50] B. Reeves and C. Nass. *The media equation: how people treat computers, television, and new media like real people and places*. Cambridge University Press, New York, NY, USA, 1996.

[51] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[52] A. Savidis. Rapidly implementing languages to compile as c++ without crafting a compiler. *Softw. Pract. Exper.*, 37(15):1577–1620, Dec. 2007.

[53] J. Scholtz and S. Wiedenbeck. Using unfamiliar programming languages: the effects on expertise. *Interacting with Computers*, 5(1):13 – 30, 1993.

[54] T. Schorsch. Cap: an automated self-assessment tool to check pascal programs for syntax, logic and style errors. *SIGCSE Bull.*, 27(1):168–172, Mar. 1995.

[55] R. L. Shackelford and A. N. Badre. Why can't smart students solve simple programming problems? *Int. J. Man-Mach. Stud.*, 38(6):985–997, June 1993.

[56] B. Shneiderman. Designing computer system messages. *Commun. ACM*, 25(9):610–611, Sept. 1982.

[57] B. Shneiderman. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[58] T. Teitelbaum and T. Reps. The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24(9):563–573, Sept. 1981.

[59] The Eclipse Foundation. http://www.eclipse.org.

[60] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.

[61] V. J. Traver. Sobre los mensajes de error de los compiladores. *Proceedings of the Actas del VII Congreso Internacional de InteraccižZ on Persona-Ordenador (InteraccižZ on 06)*, pages 345–348, Nov. 2006.

[62] V. J. Traver. On compiler error messages: what they say and what they mean. *Adv. in Hum.-Comp. Int.*, 2010:3:1–3:26, Jan. 2010.

[63] A. V. *Compilers: Principles, Techniques and Tools (Second Edition)*. Addison Wesley, 1999.

[64] M.-H. N. C. Vee, B. Meyer, and K. L. Mannock. Empirical study of novice errors and error paths. Aug. 2005.

[65] G. M. Weinberg. *The psychology of computer programming*. Van Nostrand Reinhold Co., New York, NY, USA, 1988.

[66] J. Yang. Explaining type errors by finding the source of a type conflict. In *Scottish Functional Programming Workshop'99*, pages 59–67, 1999.

[67] L. Zolman. Stlfilt: an stl error message decryptor for c++. http://www.bdsoft.com/tools/stlfilt.html.