2
Análise sobre Aprendizagem de Programação

Este capítulo discute as abordagens utilizadas para apoiar a aprendizagem de programação na graduação. Inicia com um levantamento das dificuldades no primeiro contato do aluno com a construção de programas, comparando com uma teoria de desenvolvimento cognitivo. Na Seção 2.1 são apresentadas experiências com o ensino formal de programação em cursos introdutórios de programação, destacando análises relevantes para o contexto deste trabalho. Em seguida, baseado na experiência de programação introdutória da UFAM e nos trabalhos analisados, é proposta e discutida uma forma de representação para identificar e analisar elementos ligados à evolução de códigos de alunos cursando a disciplina introdutória de programação, fornecendo pistas para se conhecer como ocorre o desenvolvimento cognitivo desses alunos em programação.

A busca por conhecimento sobre habilidades necessárias à programação é recorrente na pesquisa em computação. Em um dos clássicos sobre o assunto, Dijkstra (1982) defendeu que programar era levar o aluno ao mais alto nível de pensamento, isto é, à pura abstração. De um ponto de vista de profissionais de desenvolvimento de software, o livro de Weinberg “The Psychology of Computer Programming” (1971) aborda o aspecto psicológico da aprendizagem de programação, envolvendo, além da abstração, habilidades psicossociais. Já Knuth (2005) afirma que Ciência da Computação é “meramente” a solução de problemas, expressa através da programação. Seguindo essas considerações, aprender a programar é aprender a resolver problemas, utilizando níveis de pensamento mais elevados, requerendo maior uso da abstração. Para que isso tudo ocorra, também é necessário que haja motivação no próprio grupo social.

Embora haja muita pesquisa sobre o assunto (Weinberg, 1971) (Dijkstra, 1982), todas as etapas do raciocínio pelas quais o indivíduo passa ao adquirir as habilidades necessárias à programação ainda não são conhecidas, razão pela qual é tão difícil afirmar quais atividades ou métodos são mais adequados para tornar

mais fácil a aprendizagem de programação em cursos de graduação em computação.

Apesar das etapas de raciocínio que levam alguém a aprender programação não serem conhecidas, há muitas pesquisas em estilos, modelos e técnicas de aprendizagem que podem servir de subsídios para entender o processo de aprendizagem de programação. Jean Piaget, cujo tema central de pesquisa foi procurar entender como ocorre a aprendizagem em crianças e adolescentes, adaptou o método clínico de diagnóstico utilizado em medicina para conseguir compreender por meio de entrevistas como os indivíduos testados estavam aprendendo determinados conceitos. Seus achados foram amplamente divulgados e discutidos em seus diversos livros, como (Piaget & Inhelder, 1968), (Piaget, 1972) (Piaget, 1995).

Em (Piaget & Inhelder, 1968), Piaget estabelece estágios de aprendizagem por onde todos os indivíduos passam para elaborarem pensamentos mais sofisticados e abstratos. Esses estágios foram definidos acompanhando o processo natural do desenvolvimento infantil, servindo de parâmetro para intervenção pedagógica, que deve ser fornecida respeitando o estágio em que a criança se encontra.

Refletindo sobre os achados de Piaget e procurando estabelecer um paralelo entre esses estágios de desenvolvimento observados no desenvolvimento da criança até a adolescência, e os jovens que ingressam em cursos de graduação na UFAM que, em sua maioria, estão na fase final da adolescência. Como foi o desenvolvimento da abstração nesses alunos? Será que eles possuem dificuldades? Diferentemente do que ocorre nos primeiros estágios infantis, sensório-motor e operações concretas, os estágios seguintes vão requerendo cada vez mais operações abstratas e o refinamento destas nem sempre é exigido dos alunos no ensino médio. Isso pode ocorrer devido à metodologia adotada em muitas escolas, exigências de cumprir um currículo ou ainda pelo fato de que os alunos precisam passar em testes de avaliação fortemente baseados em conteúdos para poderem ingressar em um curso de graduação.

Como somos todos diferentes e passamos pelos estágios de desenvolvimento cada um na sua velocidade, deve-se cogitar que nem todos os alunos possuem o mesmo nível de abstração para resolver problemas. Ainda que todos consigam resolver problemas com muita destreza, para programar é

necessário abstrair essa solução, criando programas de computador, uma máquina abstrata. Se ainda examinarmos os muitos relatos na literatura como em (McKeown e Farrell, 1999), (Clancy et al, 2003), (Chamillard e Braun, 2000), (Lister e Leaney, 2003), (Eckerdal e Berglund, 2005) que afirmam que os alunos iniciantes em cursos de graduação em computação possuem alguma dificuldade em resolver problemas e possuem muito mais dificuldade na abstração das soluções, percebemos que os alunos não conseguem transferir o conhecimento adquirido nas disciplinas de conhecimentos gerais da educação básica para uma representação mais abstrata. Cientes das transições nas etapas do desenvolvimento, segundo Piaget, o processo de aquisição de níveis de abstração mais refinados dos alunos de computação deve ser trabalhado de forma semelhante, levando-os à reflexão.

As disciplinas introdutórias de programação, então, não podem se concentrar somente na dificuldade de abstração da solução. É necessário que a disciplina inclua uma etapa anterior, que deve ser fornecida seguindo algum modelo de solução de problemas para que o aluno, gradativamente consiga explicitar seus processos de raciocínio e consiga fazer a transição dessas etapas para as etapas mais abstratas de representação de programas em uma linguagem que possa ser traduzida para o computador.

Todos os trabalhos acima mencionados afirmam que a aprendizagem de métodos e a compreensão de conceitos para a construção de programas de computador não é trivial, pois requer o uso de habilidades cognitivas de alto nível e um processo de raciocínio abstrato. Alguns trabalhos, como o descrito em (Dijkstra, 1982), enfatizam que programar envolve mais raciocínio que qualquer outra habilidade. Apesar de seu caráter abstrato, programação também é uma atividade de engenharia, uma vez que seu objetivo é a produção de artefatos que precisam satisfazer requisitos de qualidade, sendo submetidos à verificação. Para se conhecer como as disciplinas introdutórias de programação identificam as necessidades e desenvolvem essas habilidades nos alunos a seção seguinte descreve alguns relatos de experiências com essas disciplinas. A Seção 2.2 descreve como, baseada na literatura revista, procuramos explicitar o processo de raciocínio dos alunos ao desenvolverem programas na disciplina introdutória.

2.1.Experiências em Disciplinas Introdutórias de Programação para Cursos de Graduação em Computação

As disciplinas introdutórias de programação são uma preocupação constante em diversas instituições de ensino de graduação (McKeown & Farrell, 1999). As causas são diversas e as soluções mais diversas ainda. O que se deve ter como parâmetros para avaliar o sucesso de uma disciplina de programação é produção de códigos ou algoritmos de forma que atendam as especificações do professor. Apesar das muitas divergências sobre métodos, técnicas, as discussões sobre que paradigma adotar primeiro, todas as referências consultadas concordam que é necessário planejar bem essas disciplinas para que a experiência do aluno seja, no mínimo, agradável e que ele consiga entender os conceitos principais.

O mercado de trabalho do egresso de cursos de computação requer um profissional capaz de planejar soluções de problemas de forma criativa, utilizando menos tempo para chegar a bons códigos. Portanto, acreditamos que a partir da primeira disciplina esses alunos dever ser estimulados com situações-problema reais e serem constantemente testados em suas habilidades de abstração e resolução de problemas. Neste sentido, em (Mckeown & Farrell, 1999), os autores recomendam que os alunos sejam encorajados a aprender técnicas de resolução de problemas já nesse primeiro curso, pois freqüentemente os alunos encontram muita dificuldade ao aplicar as competências adquiridas em sua formação básica, em disciplinas como matemática, a um novo contexto para eles que é o de programação. Isto acaba sendo uma fonte de medo e frustração, contribuindo assim para a evasão precoce nos cursos de graduação em computação. Devido a essa preocupação, em (Clancy *et al*, 2003) é descrito um esforço em desenvolver um novo formato para uma disciplina introdutória de programação baseado em sessões de laboratório. Nessa disciplina, para apoiar as atividades de programação individuais foram planejadas muitas atividades relacionadas como discussões online, exercícios de programação em pares, leitura de textos diretamente da web, anotações reflexivas, entradas no diário e colaborações usando o processo de revisões por pares para criticar as respostas dos colegas a um dado tópico.

O artigo descrito acima trata da transformação de uma metodologia envolvendo aulas teóricas e práticas para outra envolvendo somente aulas práticas,

partindo dos problemas para os conceitos, contemplando diversas atividades bem distribuídas nas sessões. Percebe-se no relato uma preocupação excessiva com o desenvolvimento de questionários para criar nos alunos o hábito da reflexão, mas metodologia se provou ineficaz para identificar a origem das dificuldades dos alunos na apreensão dos conceitos. Quanto aos exercícios de programação resolvidos em pares, não há evidências de melhoria no desempenho dos alunos como resultados dessa técnica, visto que não há registro dessas atividades.

Seguindo a linha de avaliação, o trabalho discutido em (Chamillard & Braun, 2000) descreve a combinação de algumas técnicas de avaliação em uma disciplina introdutória de programação e demonstra por análises estatísticas as diferenças e relacionamentos entre essas técnicas. O plano de curso da disciplina segue a premissa de que antes de aprenderem a programar os alunos devem estar aptos a resolver problemas. Primeiramente os alunos resolvem problemas sem o uso de uma linguagem de programação, somente aprendendo posteriormente a representar a solução em uma linguagem de programação. Nessa disciplina, após a fase inicial de resolução de problemas, as aulas prosseguem com seis sessões de laboratório onde os alunos resolvem problemas individualmente, sendo permitida a consulta a outros colegas sempre que há necessidade. É importante salientar que a complexidade dos problemas aumenta à medida que as sessões avançam. Uma vez terminada a fase individual é proposto um estudo de caso consistindo no desenvolvimento de um programa (preferencialmente um jogo) por grupos pequenos (2 a 4 integrantes). Ao final, é conduzida uma avaliação da aprendizagem comparando estatisticamente o desempenho dos alunos nas sessões de laboratório, o estudo de caso e práticas individuais controladas e sem consulta (provas) aplicada duas vezes no decorrer do período letivo.

A principal contribuição do trabalho descrito acima é a análise estatística das correlações entre os diferentes mecanismos de avaliação utilizados, embora permaneça questionável se os métodos de avaliação utilizados foram os mais adequados à aprendizagem dos alunos. Outros elementos da análise têm sua confiabilidade prejudicada devido à ausência de controle no ambiente físico de trabalho dos grupos, tornando impossível afirmar se uma dada tarefa em grupo foi desenvolvida por um único aluno, o que poderia causar um erro nas correlações.

Outro trabalho que também utiliza modelos de avaliação para apoiar a aprendizagem de programação é descrito em (Lister & Leaney, 2003). Nesse

artigo, os autores utilizam pré-avaliações dos alunos (aquelas feitas no início da disciplina ou de cada assunto) como base para categorizá-los em estágios de aprendizagem, conforme os estágios definidos na taxonomia de Bloom (Bloom, 1956) para a área cognitiva: conhecimento, compreensão, aplicação, análise, síntese e avaliação. A partir daí a disciplina é formatada de modo a oferecer atividades diferenciadas aos alunos, correspondendo aos diferentes estágios de treinamento.

Em um esforço para identificar aspectos que facilitam a aprendizagem de programação, o trabalho descrito em (Eckerdal & Berglund, 2005) afirma que através das respostas dos alunos a perguntas como “o que é programação?” é possível definir uma ordem de apresentação dos paradigmas de programação. Os autores acreditam que os alunos precisam saber o que a aprendizagem de programação realmente é, para que aprendam seus conceitos abstratos. A maioria das respostas apuradas sugere que aqueles alunos sejam primeiramente expostos a um raciocínio mais estruturado antes de serem apresentados a paradigmas mais abstratos como a orientação a objetos.

O que realmente é necessário para se aprender e facilitar a aprendizagem de programação ainda não é conhecido. Embora alguns dos trabalhos mencionados acima façam tentativas de estabelecer um roteiro comprovadamente adequado para seguir, não há trabalhos na literatura revista até janeiro de 2010 que tenham tido êxito em estabelecer métodos irrefutáveis e técnicas para aprendizagem de programação. Por outro lado, há iniciativas cujo principal objetivo é criar e manter o interesse dos alunos na disciplina utilizando abordagens inicialmente baseadas nos processos de resolução de problemas que os alunos iniciantes em cursos de graduação em computação já foram expostos durante a educação básica, necessárias à apreensão dos conceitos. Outros ainda valorizam as práticas colaborativas, como a técnica de programação em pares, parte da metodologia dos métodos ágeis de programação, como meio de envolver os alunos em projetos de interesse coletivo, proporcionando uma vivência em grupo, necessária no mercado de trabalho.

2.2. A Evolução dos Códigos em Aprendizagem de Programação

Buscar compreender quais processos cognitivos estão envolvidos na aquisição das habilidades para programação possibilita que as práticas utilizadas por cada aluno se tornem evidentes e o conhecimento gerado possa ser reutilizado em outros cursos de programação introdutória. Nesta seção apresentamos uma caracterização de diferentes maneiras de agrupar as modificações observadas na evolução dos códigos de alunos de Ciência da Computação e Engenharia da Computação cursando a disciplina de Introdução à Computação na UFAM.

As versões de códigos, objeto desta análise, foram obtidas diretamente de um banco de dados local, que foi povoado durante a realização de trabalhos práticos da disciplina introdutória, conforme planejamento e execução discutida em (Almeida, Castro & Castro, 2006).

Conforme mencionado anteriormente, na UFAM há um histórico de experiências com diferentes abordagens na disciplina introdutória dos cursos de graduação em computação. Esta disciplina vem sendo ministrada desde 1990 com o paradigma funcional e, na época em que este estudo foi realizado (2007/2008) a linguagem Haskell vinha sendo utilizada há aproximadamente quatro anos. A análise abaixo apresentada é baseada nas características da linguagem Haskell, embora os conceitos abordados sejam pertinentes a qualquer linguagem de programação.

Ao observar os códigos dos alunos, identificamos e agrupamos as modificações na evolução dos códigos em três categorias de evolução de códigos: sintáticas, semânticas e refactoring. Juntamente com exemplos em Haskell, fornecemos fragmentos em Prolog que capturam esses tipos de modificação.

Modificações sintáticas – exemplos: endentação, inserção e deleção de caracteres. Essas modificações visam tornar o código interpretado corretamente pelo interpretador Haskell, processo que sugere muitas correções por tentativa e erro na tentativa de encontrar uma solução. Tipos de modificações sintáticas:

a. Endentação – endentação da segunda linha de uma função, posicionando o código `+ x/y` após o símbolo de igualdade, para que seja visto como uma linha única pelo interpretador. O exemplo seguinte mostra um ajuste necessário (do Programa A para o Programa B) às especificidades do Haskell, sem mudança na representação do programa.

Programa A

```
f x y = x*y
      + x/y
```

Programa B

```
f x y = x*y
      + x/y
```

Uma maneira de detectar automaticamente os espaços extras acrescentados ao Programa B é transformando termos (todo caractere no programa) em elementos de lista. Feito isso, é possível comparar cada elemento da lista e inferir que tipo de modificação ocorreu. Abaixo está um exemplo de fragmento de código em Prolog, que ilustra como comparar quaisquer dois programas.

```
sameF(T1,T2) :-
      T1=.. [FunctionName|BodyList],
      T2=.. [FunctionName|BodyList].
```

b. Inserção, Modificação ou Deleção de Caractere – ex. Mal uso de , em vez de ; ou vice-versa e também a ausência de algum símbolo conectivo, como um operador aritmético. O exemplo seguinte ilustra uma alteração necessária na solução assim como um ajuste às especificidades do Haskell:

Programa A

```
f(x,y)=(x;1)+ (2,y)
```

Programa B

```
f(x,y)=(x,1)+(2,y)
```

No Programa B, foi necessário modificar o símbolo utilizado na tupla (x,1).

Uma proposta para identificação automática de tais modificações requer a implementação de uma gramática de cláusulas definidas, DCG (Definite Clause Grammar), capaz de entender códigos em Haskell. No fragmento de código abaixo há rudimentos de uma gramática em Haskell, baseada no predicado pattern/1, o qual detecta os erros sintáticos supracitados. Os outros predicados chamados expr/1, variable/1 and pattern_seq/1 são partes integrantes dos Programas A e B. Os predicados sp/0 e optsp/0 são verdadeiros (*match*) para espaços ótimos e opcionais no código.

```
decls(Z) --> pattern(X), optsp, "=", optsp, expr(Y),
          {Z=..[f,head(X),body(Y)]}.
```

```
decls(Z) --> pattern(X), optsp, "=", optsp,
          decl_seq(Y), {Z=..[f,head(X),Y]}.
decls(Z) --> variable(X1), sp, patternseq(X2), optsp,
           "=", optsp, expr(Y),  {X=..[head,X1|X2],
           Z=..[f,X,body(Y)]}.
decls(Z) --> variable(X1), sp, pattern_seq(X2), optsp,
          "=", optsp, decl_seq(Y), {X=..[head,X1|X2],
          Z=..[f,X,Y]}.
decl_seq(Z) --> declseq(X), {Z=..[body|X]}.
declseq(Z) --> expr(X), {Z=[X]}.
declseq(Z) --> expr(X), ";", declseq(Y), {Z=[X|Y]}.
```

c. Inclusão de uma nova função – ex. Acrescentar uma nova função ao programa, visando desenvolver e testar toda a solução incrementacionalmente. Isso é normalmente encontrado em códigos de alunos e indica o uso de boas práticas de programação, enfatizadas no curso introdutório da UFAM, baseadas na estratégia de divisão e conquista.

Uma maneira de detectar automaticamente tais modificações requer a implementação do mesmo tipo de regra apresentada anteriormente, uma DCG.

Modificações semânticas – exemplos: modificação nas estruturas de dados; mudança de tupla para lista; inclusão de uma função recursiva; e correção de bugs. Essas modificações afetam diretamente a avaliação da função, resultando em uma saída errada.

a. Mudança de variáveis independentes para tuplas – ex. Uma equação do Segundo grau poderia ser calculada de duas maneiras: utilizando raízes independentes ou utilizando tuplas para calcular as duas raízes ao mesmo tempo. Os dois métodos de representação são encontrados nas soluções dos alunos e estão corretos e ambos apresentam utilizam os mesmos argumentos, embora no primeiro haja uma necessidade de se duplicar a definição da função para outra variável. Exemplo:

Programa A

```
r x a b c =(-b) + e / 2*a
            where
              e=sqrt(b^2-4*a*c)
```

Programa B

```
rs a b c = (x,y)
           where
             x = ((-b) + e)/2*a
```

```
r y a b c =(-b) -  e / 2*a                    y = ((-b) - e)/2*a
          where                               e = sqrt(b^2-4*a*c)
             e=sqrt(b^2-4*a*c)
```

No Programa A duas funções são utilizadas para resolver a equação do segundo grau. Apesar do esforço extra do programador em duplicar as funções, se elas forem muito grandes a legibilidade do código seria afetada. O Programa B apresenta uma solução mais elegante e precisa para o mesmo problema. Direcionando a saída para uma tupla, no caso (x,y), e definindo localmente a fórmula o programa todo se torna mais legível, poupando o programador de um esforço extra desnecessário.

Mais uma vez, para se detectar automaticamente tais modificações uma implementação possível é utilizando a DCG. Nesse caso, a operação sobre a DCG também consiste em transformar termos em listas, mas compara cada conjunto em ambos os programas. Conforme ilustrado no fragmento de código abaixo, é possível identificar qual estrutura foi modificada entre as versões de código. A representação abaixo atende tanto estas modificações semânticas quanto as seguintes.

```
compare2(T1,T2,Subst) :-
              T1 =.. [f,H1,B1|[]],
              T2 =.. [f,H2,B2|[]],
              H1 =.. [head|Head1],
              B1 =.. [body|Body1],
              H2 =.. [head|Head2],
              B2 =.. [body|Body2],

comp2(Head1,Head2,[],Subst1),

comp2(Body1,Body2,Subst1,Subst).
```

No predicado acima, a lista de substituição foi construída, tornando possível a comparação entre T2 e T1.

b. Mudança de tuplas para listas – ex. Se uma solução pode ser representada utilizando tuplas ou listas, Segundo as noções de eficiência trabalhadas no curso introdutório, é melhor utilizar listas em vez de tuplas, no caso de se tratar de

coleções grandes. No exemplo seguinte é ilustrada a mudança de tuplas (Programa A) para listas (Programa B):

Programa A

```
composed a b = (a, b, b+a)
```

Programa B

```
composed a b = [a, b, b+a]
```

A DCG também é a base para as operações desse tipo. Nesse caso, as operações realizadas identificam diretamente as mudanças na estrutura, entre duas versões de código. Utilizando o predicado Prolog *built-in* ‘:=/2’, mudanças como as do exemplo são facilmente detectadas, conforme o fragmento de código mostrado para a modificação sintática (a).

c. Inclusão de uma função recursiva – ex. utilizando uma função recursiva em vez de uma iterativa. Na maioria dos casos encontrados, e em geral em linguagens funcionais, recursão é mais apropriada e freqüentemente leva a uma solução mais precisa. Essa é uma das principais razões pelas quais os professores têm tanta necessidade de saberem se o conceito de recursão foi plenamente entendido por seus alunos. O exemplo abaixo mostra a representação de uma solução para a função fatorial, onde o Programa A apresenta uma solução iterativa e o Programa B, uma recursiva:

Programa A

```
fat n = if n==0
        then 1
        else product [1..n]
```

Programa B

```
fat n = if n==0
        then 1
        else n * fat(n - 1)
```

Para se detectar automaticamente tais modificações deve-se implementar uma regra para identificar se um programa foi escrito utilizando uma função recursiva. Isto requer a identificação da ocorrência de termos específicos referentes ao domínio do problema presentes nas versões de código.

d. Correção de bugs – ex. pequenas modificações feitas a fórmulas ou definições de funções, podendo se caracterizar por correção simples de bugs ou em um estilo de programação por tentativa e erro. Durante o curso introdutório da UFAM os alunos têm que utilizar um método para solução de problemas antes da codificação. Nesse caso, tais modificações são um sinal de que eles podem não ter

seguido essas orientações, tornando-se uma oportunidade para o professor intervir. Sendo assim, quando uma função não funciona, os alunos ao adotarem essa prática, geralmente fazem muitas modificações consecutivas sem raciocinar sobre o problema, ignorando assim o processo de solução de problemas adotado no curso (Polya, 1986). O código abaixo ilustra a inclusão de uma estrutura condicional IF-THEN-ELSE, no caso, uma correção simples de bug:

Programa A

```
fat n = n * fat (n - 1)
```

Programa B

```
fat n = if n==0 then 1
        else n * fat(n – 1)
```

Para detectar automaticamente tal modificação deve ser implementado um predicado de casamento de padrões, o qual verifica se determinada estrutura (dependendo da estrutura que se deseja detector) é parte de uma versão seguinte. Isso também é especialmente útil para verificar se o aluno está raciocinando sobre o problema antes de iniciar a codificação.

Refactoring – exemplo: modificações cujo objetivo é melhorar o código, de acordo com métricas de qualidade conhecidas da engenharia de software. As modificações mostradas abaixo foram extraídas do catálogo de *refactorings* em Haskell (Thompson, Reinke & Li, 2006). O catálogo foi adaptado de forma que as modificações pudessem ser agrupadas em duas categorias: (i) estrutura de dados, as quais afetam a representação de dados e consequentemente todas as funções envolvidas pelo refactoring (ex. tipos algébricos ou existenciais, tipos de dados concretos ou abstratos, construtor ou função construtiva, incluir um construtor); e (ii) nomeação, a qual implica que a estrutura de ligação do programa permanece a mesma após o refactoring (ex. inclusão ou remoção de um argumento, remoção ou inclusão de uma definição, renomeação). Após verificação preliminar dos códigos dos alunos e conversas com professores que ministraram a disciplina introdutória na UFAM no período de 2004 a 2008, constatamos que as modificações do tipo (i) não ocorrem porque a declaração de tipos não é exigida desses alunos, sendo fornecida a eles quando necessário. Portanto, como esses refactorings são muito complexos para iniciantes, comentamos e exemplificamos somente aqueles que são pertinentes ao curso introdutório analisado.

a. Inclusão ou remoção de um argumento – ex. incluir um novo argumento a uma definição de função ou constante. O valor padrão do novo argumento é

fixado no mesmo nível da definição. A posição onde o novo argumento é incluído não é ao acaso: inserir o argumento no início de uma lista de argumentos implica que ele só pode ser incluído a aplicações de funções parciais. O exemplo abaixo mostra a inclusão de uma função indefinida:

Programa A	Programa B
`f x = x + 17`	`f y x = x + 17`
	`f y = undefined`
`g z = z + f x`	`g z = z + f f_y x`

Uma maneira de detectar automaticamente tal modificação é comparando duas funções e verificando se os nomes das funções ou parâmetros das funções foram modificados. A regra seguinte escrita em Prolog ilustra como identificar uma pequena modificação nos parâmetros da função, sugerindo um refinamento na solução. Ela exemplifica um casamento de padrões, mantendo os dados em uma lista substituta. O predicado apresentado abaixo é similar ao apresentado anteriormente para modificações semânticas do tipo (a), com pequenas simplificações.

```
compare1(T1,T2,Subst) :-
              T1 =.. [FunctionName|List1],
              T2 =.. [FunctionName|List2],
              comp1(List1,List2,[],Subst).
```

b. Remoção ou inclusão de uma definição – ex. exclusão de uma definição que não está mais sendo utilizada. O exemplo a seguir ilustra a remoção da tabela da função:

Programa A	Programa B
`showAll = ...`	`showAll = ...`
`format  = ...`	`format  = ...`
`table   = ...`	

Para detectar automaticamente tal modificação basta utilizar o mesmo predicado apresentado para as modificações semânticas do tipo (a), o compare2/3, que pode também verificar se o nome de uma função foi modificado ou excluído.

Desse modo, não é necessária a criação de um nome predicado e sim estabelecer os tipos de modificação que se deseja verificar.

c. Renomeação – ex. renomear um identificador de programa, o qual pode ser uma variável valorada, variável de tipo, um construtor de dados, um construtor de tipos, um nome de classe ou um nome de uma instância de classe. O exemplo abaixo ilustra a modificação na função chamada ‘fazer’ para ‘faz’:

Programa A

```
fazer = ... fazer ...
entrada  = ... fazer ...
tabela  = ...
          where
              fazer = ...
```

Programa B

```
faz    = ... faz    ...
Entrada = ... faz    ...
tabela  = ...
          where
              faz = ...
```

Utilizando-se o predicado compare/2, ilustrado nas modificações semânticas do tipo (a) é possível identificar automaticamente tal modificação. O potencial dessa modificação, além de legibilidade, é para um refactoring desejável, na detecção de plágio. Se o professor for informado de tal modificação, e verificar que se tratou de um caso isolado, notando que na versão seguinte do código daquele aluno não houve melhora em outros trechos do código quanto a métricas de qualidade de software relativas ao nível do curso, ele deve procurar um código similar nas versões de código de outros alunos, visando à detecção de plágio.

2.2.1. Identificação de Categorias na Evolução dos Códigos

As regras em Prolog e a DCG foram implementadas e testadas, conforme mencionamos anteriormente, no banco de dados com as versões de códigos dos alunos que cursaram Introdução à Computação na UFAM em 2006, ano anterior ao desenvolvimento dessa ferramenta, chamada AcKnow. O objetivo dessa ferramenta é analisar e categorizar a evolução dos códigos dos alunos, fornecendo elementos para que o professor faça intervenções no processo de aprendizagem de programação de seus alunos enquanto eles ainda estão elaborando sua solução. Como entrada de dados, o AcKnow foi projetado para receber funções presentes nos códigos dos alunos indicados por uma análise quantitativa previamente realizada por uma ferramenta de controle de versões chamada AAEP (Almeida, Castro & Castro, 2006). Essas indicações são baseadas na quantidade de versões

que cada aluno fez. Quando a essa quantidade de versões de um ou mais alunos difere significativamente da distribuição normal da turma, esse(s) aluno(s) é (são) identificado(s) como caso especial**,** ficando marcado para visualização do professor, que envia as versões diretamente para o AcKnow. Então, baseado na análise do AcKnow para cada par de versões, o professor faz uma análise mais detalhada, fornecendo feedback diretamente aos alunos. A Figura 2.1 ilustra o funcionamento recursivo do AcKnow, incluindo suas partes integrantes, como mecanismos de inferência, DCG e base de conhecimento.

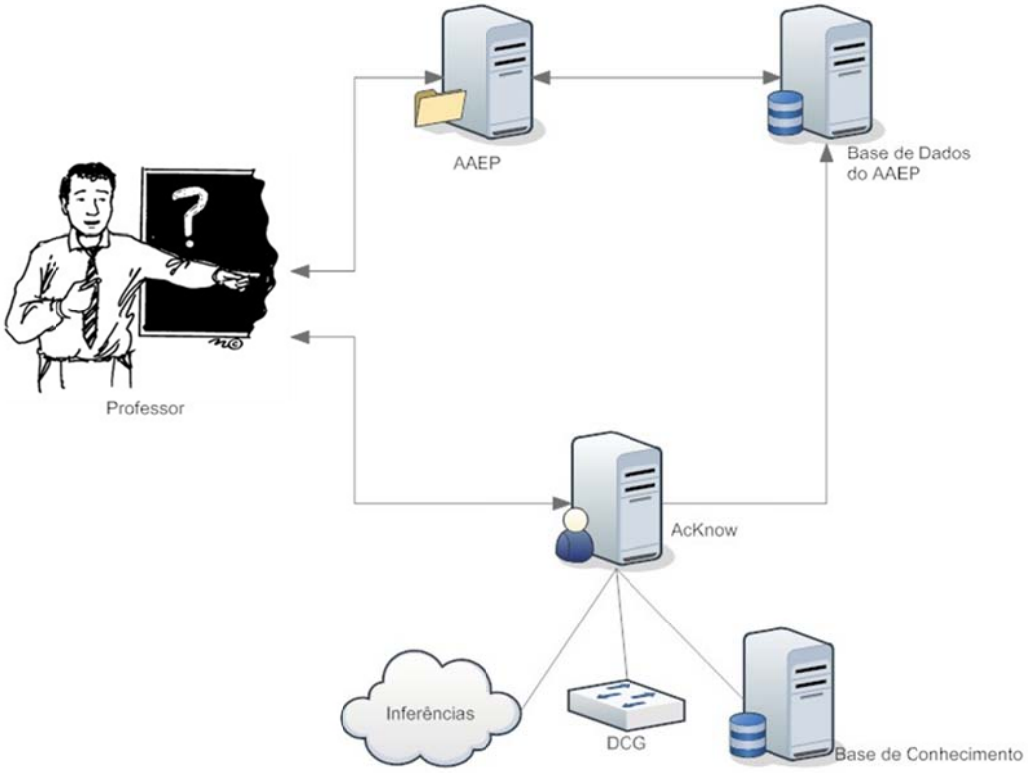

Figura 2.1 – Funcionamento recursivo do AcKnow

O AcKnow, seguindo as categorias de modificações e os exemplos apresentados anteriormente nesta seção, classifica como modificações sintáticas endentação, inclusão ou remoção de vírgula e ponto e vírgula, espacejamento entre caracteres e inclusão ou remoção de parênteses. As modificações semânticas são aquelas relacionadas a avaliação de funções. Por exemplo, modificações em estruturas de dados e operações nas funções são classificadas nessa categoria. Além disso, refactoring, neste contexto, são casos especiais de modificações semânticas cujo objetivo é a melhoria na qualidade do código segundo as métricas de qualidade de software. Atualmente o AcKnow utiliza parcialmente o catálogo de refactoring para Haskell, definido em (Thompson, Reinke & Li, 2006).

O AcKnow foi utilizado para analisar a história do desenvolvimento dos alunos da disciplina Introdução à Computação da UFAM. Como essa análise foi realizada após o fechamento da disciplina, não obtivemos o consentimento de todos os alunos para a divulgação, ainda que de forma anônima, de seus códigos. A análise foi realizada para todos os alunos da turma, sendo os resultados referentes a esta totalidade, pois obtivemos consentimento para analisar os dados, só não para mostrar os códigos da maioria. Ao analisarmos os códigos, identificamos os casos que foram apontados pelo AAEP e procuramos seus autores, obtendo seu consentimento para divulgação dos códigos de forma anônima. Dentre esses alunos, uma nos chamou mais atenção pela natureza de suas modificações, o que também divergiu dos outros alunos indicados pelo AAEP. Aqui nós a chamamos Jane Doe e iremos analisar detalhadamente a evolução de seus códigos referentes a um dos exercícios realizados após o primeiro mês do curso de quatro meses. Na Tabela 2.1 são apresentadas as categorias de modificações de Jane Doe, associadas aos intervalos de tempo entre elas.

Tabela 2.1 – Histórico da aluna Jane Doe

Versão	Intervalo	Categoria
1	0	sintática
2	mesmo minuto	sintática
3	1 minuto	sintática
4	1 minuto	sintática
5	8 minutos	refactoring
6	171 minutos	semântica
7	44 horas	refactoring

Nesta seção apresentamos a evolução dos códigos de Jane Doe como solução para o seguinte problema: Dado um segmento de reta r, que passa por dois pontos a e b, localizados no primeiro quadrante do plano cartesiano e qualquer ponto p, determine se p pertence ao segmento de reta ou à sua continuação ou se está localizado acima ou abaixo da reta.

Baseado na análise apresentada abaixo é evidenciado que Jane Doe utiliza corretamente o desenvolvimento incremental de soluções, utilizando a estratégia de divisão e conquista, perceptível em quase todas as versões de código, as quais são indícios de uma assimilação do método de solução de problemas de Polya (1986), trabalhado em sala de aula. Somado a isso, ela ainda utiliza refactoring, o

que indica uma preocupação com métricas de qualidade de software. Seguem as modificações:

(modificação 1)

```
pertseg x1 y1 x2 y2 x3 y3 = if x1>0 && y1>0 && x2>0 && y2>0
                    then if seg x1 y1 x2 y2 x3 y3
                        then "p belongs to AB"
                        else if det==0
                            then "p belongs to r"
                            else if det>0
                                then "p is above r"
                                else "p is below r"
                    else "p is not in the 1st quarter"

seg x1 y1 x2 y2 x3 y3 = (cresc x1 y1 x2 y2 x3 y3||decresc x1 y1 x2 y2 x3 y3||conth x1 y1 x2 y2 x3 y3||contv x1 y1 x2 y2 x3 y3)
```

(modificação 2)

```
det x1 y1 x2 y2 x3 y3 = x1*y2+x2*y3+x3*y1-(x3*y2)-(x2*y1)-(x1*y3)
```

Conforme evidenciados no código (modificação 1) e na (modificação 2), na primeira versão (modificação 1) ela resolve parcialmente o problema, mas a solução não é precisa o suficiente. Então, na segunda versão (modificação 1 + modificação 2), ela mantém o código anterior, acrescentando uma linha, com a descrição de uma outra função.

(modificação 3)

```
cresc x1 y1 x2 y2 x3 y3 = x3>x1 && x3<x2 && y3>y1&& y3<y2
decresc x1 y1 x2 y2 x3 y3 = x3>x1 && x3<x2 && y3<y1 && y3>y2
conth x1 y1 x2 y2 x3 y3 = x3>x1 && x3<x2 && y3==y1 && y3==y2
contv x1 y1 x2 y2 x3 y3 = x3==x1 && x3==x2 && y3>y1 && y3<y2
```

Conforme evidenciado no código acima (modificação 3), nesta terceira versão, ela acrescenta as funções necessárias para estabelecer as condições da reta.

Na quarta versão (modificação 4), ela tenta modificar a última linha do código, mas desiste após acrescentar e retirar espaços. Na quinta versão ela renomeia pertseg com segment.

(modificação 4)

segm (x1,y1) (x2,y2) (x3,y3) = (cresc (x1,y1) (x2,y2) (x3,y3)||decresc (x1,y1) (x2,y2) (x3,y3)||conth (x1,y1) (x2,y2) (x3,y3)||contv (x1,y1) (x2,y2) (x3,y3))

det (x1,y1) (x2,y2) (x3,y3) = x1*y2+x2*y3+x3*y1-(x3*y2)-(x2*y1)-(x1*y3)

Conforme mostrado o fragmento de código acima (modificação 4), após alteração no nome da função, na sexta versão, ela tem um salto cognitivo, mudando seg x1 y1 x2 y2 x3 y3 por segm (x1,y1) (x2,y2) (x3,y3) e det x1 y1 x2 y2 x3 y3 por det (x1,y1) (x2,y2) (x3,y3) porque percebeu que o problema é mais facilmente resolvido com o uso de tuplas. Na sétima versão, ela acrescenta uma nova linha no final do código.

Outra informação relevante relativa à Tabela 2.1 e à evolução do código descrita acima é que entre as versões 5 e 6 Jane Doe resolveu outros 5 exercícios da mesma lista de exercícios. Alguns desses exercícios pediam explicitamente para utilizar tuplas nas soluções. Após resolvê-los, ela retornou ao problema em questão e submeteu a versão 6 utilizando tuplas na solução. Portanto, Jane Doe teve o insight ao resolver outro problema, o que é notável em quem não tem experiência anterior com programação.

Cerca de 100 alunos estavam matriculados na disciplina de Introdução à Computação no período em que foi gerado o banco de dados com a evolução dos códigos analisados. O método estatístico utilizado pelo AAEP indicou uma média de 3 alunos por exercício como casos especiais. Fora Jane Doe, os outros alunos apresentaram um padrão parecido de modificações, com exceção das de refactoring, efetuando muitas modificações sintáticas em curto intervalo de tempo seguidas de algumas modificações semânticas intercaladas com modificações sintáticas, em intervalos maiores. Somente poucos (4 alunos em 2 exercícios diferentes) fizeram refactoring. Nesses casos em que ocorreram refactoring, eles só se concentraram nas últimas versões.

Ao analisar a evolução dos códigos dos alunos, refletindo sobre os experimentos de Piaget (1978) com crianças, percebeu-se que eles passam pelas mesmas etapas que as crianças quando estão aprendendo novos conceitos que requerem mais esforço de abstração do que sua estrutura abstrata já armazena. Primeiramente, assim como as crianças, eles tentam repetidas vezes fazer com que sua linha de raciocínio não modifique, adaptando o ambiente, no caso fazendo modificações sintáticas. Quando não conseguem por este caminho tentam modificar a estrutura do pensamento com o que já conhecem, fazendo uma ou mais modificações semânticas intercaladas com modificações sintáticas. Aqueles que já compreendem bem o problema, o que seria comparado à aquisição de habilidades cognitivas de um próximo estágio no desenvolvimento, conseguem fazer modificações de refactoring.

A identificação das sequências de utilização das categorias de evolução de códigos realizada para essa turma é uma pista importante para saber como propor atividades que incentivem o desenvolvimento do raciocínio abstrato dos alunos. Assim como afirmado por Piaget e reproduzido em (Parrat-Dayan & Tryphon, 1998) referente ao ensino básico, a aprendizagem colaborativa acelera o desenvolvimento dessas habilidades se forem incentivadas interações focadas à resolução de exercícios, uma vez que, organizados em grupos, os alunos são instigados a defender seus pontos de vista e questionar seus pares. Para que isso ocorra é essencial que haja um acompanhamento dos professores envolvidos no processo e que esses consigam identificar pistas para intervirem nas interações, sempre que houver uma dificuldade na fluidez da discussão.

2.3. Conclusão do Capítulo

Neste capítulo discutimos sobre a necessidade de abstração em programação e porque ela é um empecilho na aprendizagem de programação. Elaboramos nosso ponto de vista comparando o desenvolvimento de habilidades de programação com o desenvolvimento de habilidades cognitivas nas crianças, pois pelo que observamos, o processo é similar ao de adultos para a aquisição de habilidades de raciocínio mais abstrato, como é o caso da programação.

Descrevemos o que algumas pesquisas têm demonstrado sobre métodos e técnicas utilizados em salas de aula, com o objetivo de facilitar a aprendizagem de

programação de alunos de graduação, principalmente de cursos de computação. Com isso, identificamos que a utilização de técnicas de solução de problemas é um meio importante para sistematizar o processo de aprendizagem através da reflexão em termos de problemas, mas a escolha da linguagem de programação adotada no curso introdutório é essencial para tornar a codificação mais centrada na solução do problema e menos em recursos da linguagem.

A Seção 2.2, baseada em (Castro, Fuks & Castro, 2008b), trata da análise sobre a evolução dos códigos dos alunos de uma turma iniciante em programação. Ela explicitou, através da identificação de categorias, o processo de desenvolvimento de códigos como solução de exercícios, utilizando o método de solução de problemas de Polya (1976). Com isso, o professor, se auxiliado por mecanismos de identificação dos gargalos na evolução dos códigos, como muitas modificações sintáticas sem modificação semântica e sem êxito na solução do problema, ele pode intervir antes mesmo do aluno procurá-lo e assim contribuir para o processo de aquisição de habilidades em programação daquele aluno.

No próximo capítulo descrevemos uma investigação sobre as tecnologias que apóiam a aprendizagem de programação em grupo e ajudam o professor a intervir. Assim como o AcKnow detecta elementos do processo de aquisição do conhecimento em programação, outras ferramentas e técnicas de computação auxiliam o professor a detectar outras características necessárias à intervenção adequada.