7º seminário integrado de software e hardware

21 a 25 de julho de 1980
FEC/UNICAMP

Promoção
-SBC/Sociedade Brasileira de Computação

Patrocinio
SEI/Secretaria Especial de Informática
FEC/Faculdade de Engenharia de Campinas
- UNICAMP/Universidade Estadual de Campinas

das no texto ou estrutura do programa, quando submetidas a automação.

verificação da maior adequabilidade de uma técnica ou grupo de técnicas a uma certa área de aplicação.

1.1 Classificação dos erros estudados

O exame do processo de programação mostrou a existência de certos erros que são mais comumente encontrados em programas. Tendo em vista os objetivos apresentados para o estudo, o universo de erros cuja presença se decidiu pesquisar foi limitado a este conjunto de erros mais frequentes.

Para melhor observar o comportamento das diversas técnicas, elaborou-se uma classificação para estes erros. Assim, foram definidas duas classes principais: erros de especificação e de implementação. A primeira se restringe às falhas no texto da especificação, sem levar em conta problemas como a má comunicação entre usuário e projetista. A outra compreende todos aqueles erros não causados por especificação mal feita, dividindo-os nas subclasses:

- . codificação - causados geralmente por restrições de ordem léxica ou sintática, sendo muitas vezes apontados na fase de compilação.
- . lógica - detetados apenas durante a execução, para situações de entrada específicas.
- . desempenho - relativos à obtenção de resultados não necessariamente incorretos, de forma pouco eficiente. Tais erros (no sentido de que noções elementares de eficiência não são previstas pelo programa) manifestam-se através de problemas tais como a utilização de áreas de memória excessivamente grandes ou tempo de execução desnecessariamente longo.

Os erros de lógica sofreram também uma classificação, devido à grande variedade de erros com esta característica geral: casos, quando um conjunto de caminhos apresenta casos a mais ou a menos do que o especificado: caminhos, quando um caminho é inviável ou implementa um algorítmo incorretamente; operações com dados, englobando problemas como "overflow" e comandos de E/S e interligação de módulos, referente a problemas de passagem de parâmetros na chamada de rotinas.

Observe-se que a classificação acima não menciona erros encontrados em aplicações de tempo real, uma vez que este tipo de aplicação não foi objeto deste estudo.

1.2 Métodos selecionados para o experimento

As técnicas cujas características decidimos comparar foram as seguintes:

- execução simbólica, baseada no artigo de King [2] Arbitrou-se a cobertura do tipo "boundary- interior", já que o autor não utiliza naquele artigo nenhuma cobertura em particular;
- emprego de grafos reduzidos com cobertura mínima que utilizam caminhos entre decisões. Baseamo-nos no artigo de Huang [3]. Esta cobertura propõe a execução do menor número de caminhos possível que permita que todo subcaminho entre decisões seja percorrido ao menos uma vez.
- utilização de tabelas de condições para a formalização de especificações de programas, conforme proposta de Goodenough e Gerhart [4].

De acordo com o plano do experimento estas técnicas foram aplicadas a quatro programas, cujas especificações se encontram nos Anexos (I a IV). O emprego de várias linguagens dificultou a uniformidade da aplicação de alguns critérios, mas isto foi em muito compensado pela maior variedade de situações que puderam ser estudadas.

Os dois primeiros programas se referem a aplicações científicas:

- formatação de textos (em PL/I): um programa clássico, adotado por vários autores, havendo sido utilizada a versão de Myers [5], codificada com erros.
- emissão de folha de pagamento (em FORTRAN): trata-se de um problema comum, porém pouco mencionado na literatura especializada. Foi dada preferência ao FORTRAN em contraposição ao COBOL, em função do maior número de entraves à elaboração de casos para teste apresentado pelo FORTRAN (como áreas COMMON).
- resolução de sistemas lineares pelo método de Jacobi (em ALGOL68C). O problema permitiu o acompanhamento de operações com variáveis indexadas, sendo este um ponto crítico na aplicação de testes de programas. A linguagem escolhida é adequada para implementar este tipo de problema, oferecendo grande flexibilidade para a referência a seções de matrizes.
- pesquisa de listas encadeadas (em PASCAL). Este programa mostrou as dificuldades para teste de programas que surgem com a representação de referências a variáveis de forma indireta, através de "pointers"

2. Condução do experimento:- procedimentos e dificuldades

Um problema encontrado durante o trabalho foi o de se pesquisar uma quantidade e uma variedade de erros suficientemente grande, para permitir conclusões justificáveis, isto apesar do número relativamente reduzido de programas utilizado. O primeiro exemplo já continha erros propositalmente codificados por Myers [5]. Resolveu-se obedecer ao mesmo procedimento daquele autor, semeando-se erros nos demais.

Tal decisão implica na obtenção de resultados parcialmente viciados, por mais isenta que seja a aplicação dos métodos de testes. No entanto, não se buscou apenas resultados quantitativos (como % de erros descobertos), mas principalmente qualitativos (tais como, os erros mais sensíveis à detenção através de certo tipo de enfoque). Optando-se pela introdução proposital de erros, a dificuldade natural de construir-se programas confiáveis fez com que o aspecto quantitativo não fosse muito prejudicado, já que de todos os erros finalmente computados, 41% não eram conhecidos antes do início dos testes.

A análise de ferramentas automatizadas foi realizada através da simulação manual dos processos de automação. Para tal, tornou-se necessário criar variáveis e vetores auxiliares, de forma semelhante à automação real, para permitir passagem de parâmetros entre rotinas, referências a índices de vetores e matrizes, além do armazenamento de sequências históricas de criação e utilização de "pointers". Muitos comandos, principalmente os de ALGOL, devido à grande flexibilidade permitida pela linguagem, exigiram o desdobramento de comandos em comandos de atribuição e desvio equivalente.

Cabe destaque ao tratamento dado às chamadas de rotinas, encontradas nos dois primeiros programas. Das muitas soluções recomendadas para este tipo de problema, duas foram adotadas visando comparação posterior. No primeiro programa, cada comando CALL foi substituido, quando do desenvolvimento do teste, pelo bloco da rotina ativada. No segundo, efetuou-se análise preliminar das rotinas, de forma a obter os pares (predicado, saída) correspondentes à cobertura desejada; cada comando CALL foi então substituido pelos diversos pares assim deduzidos.

A aplicação das técnicas nos quatro programas obedeceu a uma ordem fixa. Construiu-se inicialmente a tabela de condições, porque esta pressupõe desconhecimento do interior do programa. Executou-se, em seguida, as duas outras técnicas, em ordem crescente de complexidade de cobertura. Esta ordenação foi feita com o intuito que houvesse a menor interdependência possível entre as técnicas estudadas.

3. Resultados e conclusões

3.1 Descoberta de erros

a) Fases de descoberta

Durante a aplicação das técnicas de teste, foi possível distinguir-se a existência de três etapas principais no processo de descoberta de erros. A primeira ocorre quando se utiliza as técnicas para definição de dados para teste: nesta fase, as técni-

cas baseadas na estrutura do programa detetam erros de implementação (por exemplo, caminhos inviáveis) e as baseadas na especificação apontam incoerências na definição do problema, ao analisá-la para elaboração da tabela. A segunda etapa é característica do enfoque baseado na estrutura: de posse dos pares (predicado, saída) o cotejo desses com a especificação permite a descoberta de erros de implementação nos caminhos executados. A etapa final corresponde à deteção de erros no programa quando este é executado utilizando-se os dados para teste elaborados na primeira fase: ambos os enfoques permitem apontar erros de implementação.

b) Resultados numéricos

A tabela do anexo V mostra a distribuição dos 86 erros descobertos, indicando a contribuição de cada técnica na sua deteção. Convencionou-se definir descoberta "fortuita" como aquela em que uma técnica aponta um erro apenas por sorte, quer pela escolha acidental de valores que o evidenciem (por exemplo, na satisfação de quesitos das tabelas), quer pela execução de caminhos não necessariamente obrigatórios para satisfação da cobertura estabelecida (enfoque baseado na estrutura interna). Assim é que a técnica de cobertura mínima de caminhos entre decisões evidenciou 66% dos erros, dos quais 8% de forma fortuita; a de enfoque externo apontou 69% dos erros, sendo que 6% fortuitamente; e a execução simbólica, que melhor desempenho apresentou, detetou 86% dos erros descobertos, com apenas 3% de ocorrências fortuitas.

Conforme mencionado anteriormente, 41% do total de erros relacionados eram ignorados previamente. Destes, a maior parte se referiu a erros de algorítmo (25%) e situações não especificadas (16%), além de erros de desempenho (8%) e codificação (8%). A técnica que mais influência exerceu na descoberta destes erros desconhecidos foi novamente a execução simbólica, com a de tabelas de condição ficando em segundo lugar.

O quadro permite também observar que alguns tipos de erros (como os de interligação entre módulos), por sua ocorrência reduzida, não possibilitaram elaboração de conclusões definitivas sobre a atuação de cada técnica para sua descoberta. Os erros de especificação, conforme se supôs inicialmente, foram apontados com maior frequência pela técnica das tabelas de condições, enquanto as técnicas baseadas na estrutura do programa se mostraram mais adequadas para colocar em destaque erros como os de desempenho e de falhas em caminhos, que exigem análise de expressões de caminhos.

Note-se que não se pode garantir a existência de apenas 86 erros nos programas estudados. Afirma-se, apenas, que se conseguiu descobrir 86 erros.

c) Tipos de erro descobertos por fase

Na primeira das fases de descoberta de erros, referente à aplicação das técnicas para obtenção de dados para teste, o método baseado na especificação constatou a existência de incoerências nas definições do problema. Os outros dois métodos isolaram erros de codificação (pela análise do fluxo de dados, especialmente na execução simbólica), caminhos inviáveis e infinitos, operações incorretas com dados e problemas de desempenho (ao analisar o conjunto de caminhos executados, com a verificação da existência de comandos em excesso).

A fase de comparação dos pares (predicado, saída) com a especificação permitiu apontar erros de algorítmo (quando um determinado conjunto de condições - o predicado - não causava uma saída de acordo com o especificado). Foi possível também encontrar casos em excesso, por haver pares (predicado, saída) sem correspondência na especificação. Note-se que não se tentou, aí, analisar o enunciado do problema de forma a verificar se todos os seus quesitos estavam sendo cumpridos, mas apenas se os caminhos executados conforme a cobertura pré-estabelecida correspondiam ao desejado.

Na fase final, de execução dos programas, vários erros foram evidenciados, por ambos os enfoques. Só nesta etapa os resultados da aplicação de testes a partir da tabela apontaram erros de implementação por esta diferir do previsto para uma determinada situação (erros de algoritmo e casos a menos, principalmente). O enfoque baseado na estrutura do programa indicou erros devidos a existência de valores inesperados na saída, na maior parte das vezes causada por algorítmos incorretamente implementados, operações com dados e chamadas inválidas de rotinas.

3.2 Uso de ferramentas automatizadas

Embora não fossem diretamente empregadas ferramentas automatizadas neste trabalho, o funcionamento das mesmas foi simulado manualmente, foi possível concluir-se alguns pontos sobre a viabilidade de seu uso e utilidade.

A principal vantagem deste tipo de procedimento é a de eliminar tarefas repetitivas, poupando tal trabalho ao aplicador de testes. A execução de caminhos é bastante extenuante e muitas vezes redundante, pois vários caminhos definidos por uma cobertura diferem apenas em uns poucos comandos finais.

A automação mostrou-se igualmente indicada para atividades "contábeis", como totalizar e apontar comandos executados, blocos e rotinas ativados, permitindo observação da eficiência da cobertura adotada. Este tipo de técnica pode igualmente ser de utilidade em operações de análise de fluxo de dados, onde se verifica a variação de valores de cada variável ao longo de um conjunto de caminhos, permitindo a constatação de incoerências em atribuições ou presença de variáveis nunca utilizadas.

Uma das desvantagens encontradas é a sua dependência da linguagem de codificação. Para cada tipo de programa, principalmente o codificado em ALGOL, foi necessário pré-estabelecer um conjunto de convenções de transformação de comandos compostos em comandos primários, para ser possível executar "automaticamente" os caminhos. Nota-se, desta forma, a inviabilidade de proposições para o desenvolvimento de geradores universais de dados para teste.

Outro problema com os métodos automáticos diz respeito ao crescimento das áreas de armazenamento utilizadas (e o tempo consumido) para a execução dos caminhos, à medida em que a estrutura analisada cresce em complexidade. Programas de médio ou grande porte certamente gastarão muito tempo e muita memória para serem testados com a assistência de métodos automáticos.

A maioria das ferramentas automatizadas se propõem a gerar dados necessários ao teste dos programas, utilizando o próprio programa como entrada. Verificou-se que, na verdade, só se pode utilizá-las com alguma esperança de êxito, em apenas duas situações especiais. Na primeira, define-se uma classe de dados e a ferramenta "segue" o programa indicando quais os caminhos percorridos e, possivelmente, o formato geral dos resultados. Na segunda, mais complexa, escolhe-se um grupo de caminhos a executar e obtém-se a forma geral dos dados necessários a esta execução. Esta última situação é bastante difícil, na prática, a menos que seja aplicada apenas em uns poucos casos especiais.

A geração automática de todos os dados necessários para teste é, segundo nossa observação, uma meta longe de tornar-se real.

3.3 A necessidade de intervenção do programador no processo de teste

No decorrer deste trabalho, foram muitas vezes usadas expressões como "escolha de caminhos", "definição de critério de cobertura", "análise da especificação". Estes e outros procedimentos, essenciais à atividade de teste, só podem ser levados a cabo através da interação constante homem-máquina (ou, em outro nível, homem-programa testado). Este é mais um argumento a ser utilizado em resposta aos defensores de ferramentas de teste totalmente automatizadas.

Deve-se sempre ter em mente que a atividade de teste é essencialmente dinâmica. Resultados obtidos de uma determinada execução podem vir a anular todos os anteriores, exigindo reinício do processo de geração de dados para teste. Recomenda-se, assim, que cada resultado de execução seja analisado, antes de se dar proceguimento à sequência planejada de testes, evitando desperdício de esforços.

3.4 Tratamento de rotinas

A substituição de comandos de chamada pelo corpo da rotina revelou-se de aplicação mais simples, por imediata. Apresentou, no entanto, o inconveniente de aumentar significativamente o número de caminhos a percorrer, muitos dos quais se mostraram inviáveis. Há assim um acréscimo de nós a serem visitados, com consequente gasto de tempo (do aplicador de testes e da automação, quando empregada).

A outra alternativa examinada, de troca das chamadas por todos os pares (predicado, saída) admissíveis para uma cobertura restringiu o número de caminhos a percorrer. Conseguiu-se, inclusive, eliminar de imediato várias opções inviáveis, com consequente economia de tempo de pesquisa. Este procedimento apresenta grandes vantagens quando a chamada de rotinas é feita utilizando-se valores ao invés de variáveis. Em tal caso, o resultado é obtido pela simples substituição dos mesmos valores na expressão de saída apropriada.

Esta última tática apresenta, no entanto, o inconveniente de exigir um trabalho inicial de síntese das rotinas ativadas. Deve-se também ressaltar que, nem sempre, as expressões obtidas correspondem a uma redução compensadora. Quando a rotina é linear, sem muitos ciclos ou desvios, o tratamento equivale à simples substituição. Para rotinas com um número excessivo de testes e ciclos, o número de pares obtidos é tão grande que acarreta novos problemas para a execução dos caminhos. Nesta última hipótese, nenhum dos dois métodos empregados é de grande utilidade, devendo-se optar por outras formas de tratamento.

3.5 Sensibilidade ao tipo de aplicação

O enfoque de teste baseado na especificação (tabelas de decisão) foi mais eficiente em termos da relação tempo dispendido versus resultados obtidos, no caso das aplicações comerciais, já que as mesmas muitas vezes descem a um grande nível de detalhe e subentendem caminhos que dificilmente serão atingidos por coberturas como as utilizadas neste trabalho.

As aplicações científicas estudadas, no entanto, foram melhor exploradas pelas técnicas baseadas na estrutura interna dos programas por ser possível exprimir resultados sob a forma de expressões,as quais por sua vez estão explícitas ou implícitas na especificação (como no método de Jacobi). Além disso, o domínio de entrada, extremamente vasto, impediu sua análise adequada pela técnica das tabelas (no programa mencionado, por exemplo, englobava todas as matrizes reais NxM). A execução simbólica foi a mais bem sucedida neste tipo de aplicação, exatamente pela forma de apresentação dos resultados previstos.

3.6 Comparação das classes de métodos

O método de teste baseado nas especificações mostrou sua utilidade na análise da completeza e da coerência da definição do problema. A necessidade do exame detalhado dos ítens especificados permite a indicação da presença de falhas nos mesmos.Uma característica pouco ressaltada é a de que, algumas vezes, condições explicitadas na tabela de testes correspondem a caminhos especiais de um programa, que só são atingidos por coberturas extremamente rígidas. Este é o caso de uma coluna que pede a iteração de um ciclo 500 vezes, por correspondência a uma restrição da especificação.

O teste baseado na estrutura do programa, por sua vez, apresentou resultados mais positivos. O conhecimento, ainda que parcial, do interior do programa serve de auxílio às fases posteriores de desenvolvimento e manutenção do "software". Permite, principalmente, que muitos erros sejam descobertos e corrigidos mais cedo, evitando que se atinja a etapa de execução dos testes com muitas falhas no programa.

Uma característica marcante deste enfoque é exigir um desenvolvimento sistemático de casos para teste. A escolha de uma cobertura, a seleção de caminhos (com opção para redução de grafos) e sua execução passo a passo evitam o surgimento de muitos erros na própria tarefa de aplicação de testes.

Saliente-se, ainda, o fato de que há falhas - como as de desempenho - que não causam necessariamente resultados errados, só podendo ser apontadas pela análise de conjuntos de caminhos.

Pode-se afirmar que a grande desvantagem dos métodos estruturais é que eles praticamente ignoram a especificação.

Tais métodos apresentam, também, o problema de dependência da cobertura: quanto mais rígida esta última, maior o nível de segurança e maior o número de caminhos a percorrer. A decisão quanto à escolha do nível de cobertura é por isso bastante crítica: um nível muito estrito, que exija a execução de muitos caminhos, pode ser inviável por implicar um tempo irreal para a aplicação dos testes.

Um grave defeito do método baseado nas especificações está no fato que sua aplicação é realizada de forma pouco sistemática. A tabela é construida pelo aplicador de testes à medida em que condições a serem testadas lhe vêm à mente, isto é, enquanto ele analisa a especificação. O grau de sucesso deste tipo de procedimento, por este motivo, depende ainda mais da habilidade e inspiração do criador da tabela que no outro enfoque.

Apresenta ainda a desvantagem de, ignorando o interior do programa, omitir testes de condições errôneas ou casos excessivos que podem ter sido incorporados ao código à revelia da especificação. Dificilmente permite descoberta de erros como ciclos infinitos ou acesso a áreas indevidas da memória.

O próprio fato de omitir a análise da lógica faz com que, muitas vezes, defina um número de casos desnecessariamente grande, que poderia ser reduzido pelo exame - ainda que superficial - do programa. Um exemplo disto ocorreu quando da elaboração de testes para o 4º programa: a técnica de enfoque externo definiu como necessária a utilização de 190 conjuntos de dados para teste; a execução simbólica, entretanto, com apenas 9 casos, conseguiu descobrir um maior número de erros.

3.7 Comparação entre as técnicas

a) Tabelas de decisão e utilização de grafos reduzidos com cobertura mínima

Os resultados obtidos mostraram que estas duas técnicas se equivaleram quantitativamente, evidenciando número semelhante de erros - 66% e 69% do total de erros considerado. Isto, no entanto, não significa que apresentassem resultados qualitativamente iguais, pois estes erros não foram os mesmos.

É possível, entretanto, deduzir deste fato que o uso de cobertura mínima segundo foi adotado neste trabalho corresponde de certa forma a teste de programas a partir da construção de tabelas.

b) Execução simbólica e utilização de grafos reduzidos com cobertura mínima

A execução simbólica, técnica que melhores resultados apresentou, mostrou-se em muito superior à técnica de grafos reduzidos. As duas principais razões foram a cobertura mais rígida utilizada e a análise dos caminhos comando a comando, com representação funcional das saídas, de grande utilidade nas aplicações científicas.

A redução de grafos, embora simplificadora,faz com que sejam perdidos alguns detalhes na passagem de comando a comando, dificultando, por exemplo, a análise do fluxo de dados.

Observe-se, também, que as coberturas utilizadas embora pouco rígidas, proporcionaram mesmo assim, resultados satisfatórios. A comparação entre a execução simbólica e a técnica de tabelas de decisão foi feita implicitamente nos itens (a) e (b) desta seção.

c) Conjunto de técnicas analisado

Das três técnicas, a melhor foi, sem dúvida, a execução simbólica. É preciso, no entanto, ressalvar que sua aplicação é muitas vezes pouco prática em programas de grande porte, devido aos custos (tempo e máquina) que ela implica.

Se considerado o fator custo-benefício em tais programas em muitos casos a técnica mais adequada, dentre as três estudadas, é a de elaboração de casos-teste a partir da especificação. De qualquer forma, as desvantagens de se adotar um único enfoque já foram mencionadas, recomendando-se, assim, o emprego

de ambos para a obtenção de melhores resultados.

3.8 A obrigatoriedade de execução

A fase de execução dos programas utilizando os dados definidos para teste é obrigatória. Se esta afirmativa é óbvia quando se adota um método baseado na especificação, ela não é tão evidente nos casos em que o teste é baseado na estrutura interna de programas.

É comum a alegação de que o conjunto de pares (predicado, saída), aliado às observações feitas durante a execução dos caminhos associados a eles, é suficiente para indicar erros no programa, se esses erros existirem (e a cobertura for adequada). Ocorre, no entanto, que as restrições impostas por compiladores ou pelo ambiente de execução podem, muitas vezes, invalidar as expressões relativas à execução dos caminhos. A utilização "real" dos dados de teste permite constatar a validade de tais expressões - erros de precisão, por exemplo, dificilmente são detetados, a não ser por execução dos testes. Uma prova disto é que um dos erros anotados e que não foi detetado por técnica alguma referiu-se a precisão insuficiente de valores de saída, para certo tipo de entrada.

Esta observação é especialmente dirigida aos adeptos da execução simbólica, em geral atraídos por sua forma de apresentação das saídas. Teoricamente, bastaria uma simples substituição dos símbolos (entradas) por valores do domínio de entrada para se obter o resultado real. Na prática, isto nem sempre é verdade. O mesmo tipo de discussão é válido para mostrar falhas de procedimentos na prova formal de programas, da qual a execução simbólica aproveitou muitos conceitos. Verificação formal de programas pressupõe ambiente ideal de execução, dispensando a execução real. Por esta razão já foram encontrados erros em programas provados corretos formalmente.

ANEXO I

Formatar um texto lido de forma que:

1) cada linha formatada tenha no máximo 20 caracteres;
2) nenhuma palavra seja dividida em duas ou mais linha;
3) cada linha contenha o maior número de caracteres possível.

O texto de entrada está contido em cartões de 80 posições (um texto por cartão) sendo que seus caracteres são classificados como caracteres especiais. Um caracter especial pode ser um branco, um indicador de mudança de linha (&) ou de fim de texto (/). Indicadores de mudança de linha devem ser tratados como brancos pelo programa. Os caracteres '&' e '/' não devem aparecer no texto formatado.

Uma palavra é caracterizada por sequência de caracteres não especiais, sendo as palavras separadas entre si por caracteres separadores. Um caracter separador é uma sequência de um ou mais caracteres especiais. Qualquer caracter separador deve ser transformado em um único branco na saída.

Se o texto de entrada contiver uma palavra que seja grande demais para caber em uma única linha, o programa deve emitir mensagem de erro e terminar. Se o cartão lido não contiver caracter de fim de texto (/) o programa deve emitir mensagem indicando este erro e tentar ler o cartão sequinte.

ANEXO II

Elaborar programa para emissão de folha de pagamento de uma empresa, a partir dos arquivos: cadastro de funcionários e arquivo de cartões com dados relativos ao aproveitamento mensal do funcionário. O conteúdo dos registros do cadastro e dos cartões é o seguinte:

a) Cadastro

- matrícula do funcionário (inteiro, 5 casas)
- nome (40 posições)
- salário-hora (real, 5 dígitos, com 2 casas decimais)
- mês e ano de admissão (inteiros, 2 dígitos cada um)
- horas de trabalho especificadas no contrato (3 dígitos)
- número de dependentes (inteiro, dois dígitos)
- departamento (inteiro, 2 dígitos, variando de 01 a 10)

b) Cartão com variáveis relativas ao mês

- matrícula
- nome
- horas trabalhadas (real, 5 dígitos, 2 casas decimais)
- horas abonadas (real, 5 dígitos, 2 casas decimais)
- descontos (real, 6 dígitos, duas casas decimais)

O programa deve inicialmente ler conjunto de cartões-mestre, contendo

- número de feriados do mês, mês e ano processados (inteiros, dois dígitos cada)
- tabela progressiva do imposta de renta - um cartão de faixa, sendo que cada um contém
  . teto salarial (8 dígitos)
  . desconto (real, 5 casas, 2 decimais)
  . alíquota a deduzir (real, 5 dígitos, 2 decimais)
- salário por dependente (4 dígitos)

A folha de pagamento emitida deve conter um relatório de funcionários, seguido de um relatório de departamentos, com as seguintes indicações:

a) Relatório de funcionários

Uma linha por funcionário, contendo sua matrícula, nome, salário bruto, descontos de INPS, IR e outros e o salário líquido.

Na linha seguinte, caso o salário-hora do funcionário tenha sofrido reajuste por fazer jus a quinquênio, a mensagem: "INÍCIO DE QUINQUÊNIO. NOVO SALÁRIO HORA E" - seguido do novo salário-hora.

Casos descontos excessivos impeçam o pagamento, imprimir apenas a matrícula e o nome, além da mensagem: "LÍQUIDO INSUFICIENTE".

Deve ser impresso um máximo de 10 funcionários por página.

b) Relatório de departamentos

Imprimir uma linha por departamento, com os campos: número do departamento, totais de (bruto, líquido, INPS, IR) e salário líquido médio do departamento. Deve ser impresso um máximo de 5 departamentos por página.

c) Totais gerais

Ao fim da emissão dos relatórios acima, imprimir uma linha adicional, em página separada, contendo os totais relativos a (bruto, líquido, média de líquidos, INPS e IR) para toda a empresa, além do total de funcionários que apresentarem líquido insuficiente.

Os cartões de entrada com dados sobre cada funcionário já estão classificados na mesma ordem dos registros do cadastro (ordem crescente de número de matrícula) e sofreram processo anterior de crítica. Caso se verifique que esta ordem foi desobedecida (cartão e registro com matrícula diferentes) o programa deve terminar, imprimindo a mensagem: "MATRICULAS DESCASADAS".

Cálculo dos valores desejados:

. Salário bruto

Bruto = sal.hora * (horas trabalhadas)/(horas contratuais) + num.dependentes * salário por dependente

. Horas efetivamente trabalhadas

horas efet.trab = horas trabalhadas + horas abonadas + horas feriados

. Horas feriados

horas feriados = 8 * num. feriados

. Descontos (sobre o bruto):

INPS = 8% do bruto
Outros = retirado do cartão do funcionário (correspondem a empréstimos, etc)
IR = aplicar a tabela progressiva ao valor (BRUTO--INPS-OUTROS)

Salário Líquido

líquido = bruto - descontos

O funcionário deve ter seu salário-hora aumentado em 5% ao primeiro quinquênio, 10% ao segundo e 15% ao terceiro, contados a partir da data de contratação. A mensagem a respeito é emitida apenas no primeiro mês de cada quinquênio, para permitir reajuste do cadastro. Desta forma, o salário-hora nos meses subsequentes dispensa esta correção.

ANEXO III

Elaborar um programa para calcular e imprimir a solução de sistemas lineares utilizando o método de Jacobi.

Os dados de entrada, perfurados em cartões, são:

- número de equações N (inteiro, maior ou igual a 1)
- matriz A dos coeficientes (dimensão NxN)
- matriz B dos termos independentes (dimensão 1xN)

Algoritmo de Jacobi

Dado o sistema escrito sob forma matricial:

$$\begin{bmatrix} A_{11} & \cdots & A_{1N} \\ & & \\ A_{N1} & \cdots & A_{NN} \end{bmatrix} \cdot \begin{bmatrix} X_1 \\ \vdots \\ X_N \end{bmatrix} = \begin{bmatrix} B_1 \\ \vdots \\ B_N \end{bmatrix}$$ então

$$(I) \quad X_i = \frac{1}{A_{ii}} \left(B_i - \sum_{j=1}^{i-1} A_{ij}X_j - \sum_{j=i+1}^{N} A_{ij}X_j \right)$$

Numa primeira aproximação, considerar $X_i = B_i/A_{ii}$. A aplicação deste valor em (I) fornece a aproximação seguinte, e assim por diante, até obter o resultado final. Este é atingido quando a diferença entre duas aproximações sucessivas for menor que um determinado valor de tolerância. Considerar, para efeito de programa, uma tolerância de 0.005.

OBS: As matrizes A de entrada não dão origem a sistemas indeterminados. Desta forma, mesmo que contenham um ou mais zeros na diagonal principal, sempre se poderá obter uma matriz equivalente E (através de troca de linhas e colunas) tal que $E_{ii} \neq 0$ para qualquer i.

ANEXO IV

Elaborar programa que lê duas sequências de números reais e informa quantas vezes cada número da 2ª sequência ocorre na primeira. Os valores se encontram perfurados em cartões e a primeira sequência vem separada da segunda pelo número -77.0. Para cada número da 2ª sequência não contido na 1ª, o programa deve imprimir mensagem indicando o fato.

EX:

O conjunto de valores lidos:

9.0 -8.7 3.2 4.5 10.0 -77.0 3.2 4.5 4.5 6.0

deve dar origem ao seguinte tipo de saída:

valor	nº ocorrências
3.2	1
4.5	2

O valor 6.0 não existe na 1ª sequência.

ANEXO V

TABELA DE INCIDÊNCIA DE ERROS E SUA DESCOBERTA POR TÉCNICA

TIPO DE ERRO	TÉCNICA								
	CONSTRUÇÃO DE TABELAS			GRAFOS REDUZIDOS			EXECUÇÃO SIMBÓLICA		
	DESCOBERTOS			DESCOBERTOS			DESCOBERTOS		
	SIM		NÃO	SIM		NÃO	SIM		NÃO
	F	NF		F	NF		F	NF	
ESPEC. DÚBIA		1	1			2			2
ESPEC. OMISSA		4	1		2	3		2	3
CODIFICAÇÃO		4	3	2	4	1	2	5	
CASOS EM EXCESSO		2	2	1	2	1		4	
CASOS A MENOS	1	7	4	2	4	6		10	2
ALGORITMO	3	20	1	1	18	5		22	2
CAMINHOS (*)			3	1		2		3	
OPER.: TIPOS	1	4			5			5	
OPER.:E/S		6	2		6	2	1	6	1
OPER.:MEMÓRIA		3	2		2	3		4	1
INTERL. MÓDULOS		2			2			2	
DESEMPENHO		1	8		5	4		8	1
TOTAL	5	54	27	7	50	29	3	71	12

(*) Referente a término de caminhos (caminhos inviáveis ou infinitos)

F - a descoberta do erro se deu de forma fortuita

NF - o erro não foi descoberto fortuitamente.

REFERÊNCIAS

[1] Medeiros, C., "Estudo comparativo de técnicas de teste de programas", dissertação de Mestrado, Deptº de Informática, PUC/RJ, dez. 1979.

[2] King, J.C. "Symbolic execution and program testing", Comm. of ACM, vol. 1, nº 7, jul. 1976, pg. 385-391.

[3] Huang, J.C. "An approach to program testing", Computing Surveys, vol. 7, nº 3, set. 1975, pg. 113-128.

[4] Goodenough, J.B.; Gerhart, S.L. "Toward a theory of testing data selection criteria", in Current Trends in Programming Methodology, (Yeh ed.), vol. 2, Prentice-Hall, 1977, pg 44-80.

[5] Myers, G.J., "The art of software testing", John Wiley & Sons, N.York, 1979.

CURRICULA VITAE

Cláudia M.B. Medeiros - M.Sc. - PUC/RJ; Analista de Sistemas das Centrais Elétricas de Furnas.

Carlos J.P. Lucena - Ph.D. - UCLA; Professor Associado do Departamento de Informática da PUC/RJ.

Expand: Uma Linguagem Expansível através de macros sintáticas para minicomputador.

Autores: Beatriz Zakimi
Paulo Cesar Kullock
José Lucas Rangel*

Apresentador: Beatriz Zakimi

Entidade: COPPE/UFRJ

RESUMO

Uma proposta de Leavenworth |1| Cheatham |2| e outros autores, introduziu na literatura o conceito de macro sintática como um poderoso mecanismo de extensão para linguagens de alto nível. Semelhantemente às macros utilizadas em assembler, construções da linguagem básica frequentemente utilizadas podem receber um nome, o que permite escrever programas de forma mais concentrada e legível, ao custo de mais uma fase na compilação: a da expansão das macros. Ao contrário das macros utilizadas em assembler, as macros sintáticas permitem parâmetros mais gerais, não necessáriamente restritos a um identificador, e permitem também que um formato mais livre seja utilizado, usando-se símbolos ou palavras-chave que aumentem a legibilidade. Esses parâmetros são basicamente determinados pela categoria sintática a que pertencem, e que deve ser previamente explicitada.

*Prof. Assistente, Instituto de Matemática da UFRJ

Este trabalho descreve a implementação realizada no computador MITRA-15 do Laboratório de Automação e Simulação de Sistemas do Programa de Engenharia de Sistemas e Computação da Coordenação de Programas de Pós Graduação em Engenharia da UFRJ, de uma linguagem por nós denominada EXPAND, baseada na proposta feita por Aho-Ullman |3|. A sintaxe da linguagem está na Seção I. A implementação em um minicomputador procura permitir que facilidades normalmente só encontradas em compiladores de linguagem da alto nível para máquinas maiores fiquem disponíveis, através da sua definição por macros, em minicomputador.

I, A linguagem-base

Uma vez que facilidades mais poderosas devem ser obtidas através de macros, a linguagem base contém apenas aquelas facilidades consideradas essenciais para garantir a generalidade do uso da linguagem. O leitor atento notará que algumas das construções apresentadas poderiam ser dispensadas, e simuladas através de macros. Essas construções foram incluídas na linguagem base nos casos em que a simulação através de macro seria longa, tediosa e/ou ineficiente. Entre elas se incluem mais notadamente o bloco tipo função e os procedimentos.

A sintaxe da linguagem base vem descrita a seguir; o compilador da linguagem base está descrito na seção III.

```
<bloco>::= BEGIN <lista declarações>;
           <lista comandos> END
<lista declarações>::= <lista declarações>;
                       <declaração>
                     | <declaração>
<declaração>::= <decl simples>
              |<decl>
              |<decl procedure>
<decl simples>::= INTERGER <lista identificadores>
                | REAL <lista identificadores>
<decl>::= LABEL <lista identificadores>
        | INTERGER VECTOR <lista vetores>
        | REAL VECTOR <lista vetores>
<lista identificadores>::= <lista identificadores>,
                           <identificador>
                         | <identificador>
<lista vetores>::= <lista vetores>, <vetor>
                 | <vetor>
<vetor>::= <identificador> [<expressão>:<expressão>]
<decl procedure>::= <rotina>
                  | <função>
<rotina>::= PROCEDURE <cabeçalho> <bloco>
          | PROCEDURE <cabeçalho>
            <comando composto>
<função>::= INTEGER PROCEDURE <cabeçalho>
            <fbloco>
          | REAL PROCEDURE <cabeçalho>
            <fbloco>
<cabeçalho>::= <identificador>; <parametros>
             | <identificador>;
<parametros>::= (<lista identificadores>);
                <lista decl simples>
<lista decl simples>::= <lista decl simples>;
                        <decl simples>
                      | <decl simples>
<fbloco>::= FBEGIN <lista declarações>;
                   <lista comandos> FEND
<lista comandos>::= <lista comandos>;
                    <comando>
                  | <comando>
<comando>::= <lista rotulos>:<comando sem rótulo>
           | <comando sem rótulo>

<lista rotulos>::= <lista rótulos>:<rótulo>
                 | <rótulo>
<rótulo>::= <identificador>
<comando sem rótulo>::= <bloco>
                      | <comando composto>
                      | <comando incondicional>
                      | <comando condicional>
                      | <comando atribuição>
                      | <comando result>
                      | <chamada>
                      | <comando de E|S>
<comando composto>::= BEGIN <lista comandos> END
<comando incondicional>::= GOTO <rótulo>
                         | GO <rótulo>
                         | GO TO <rótulo>
<comando condicional>::= IF <condição> THEN
                         <comando>
<condição>::= <expressão> <relação> <expressão>
<relação>::= GE|GT|LE|LT|EQ|NE
<comando de atribuição>::= <lista variaveis>:=
                           <lista expressões>
<lista de variaveis>::= <lista de variaveis>,
                        <variavel>
                      | <variavel>
<lista de expressões>::= <lista de expressões>,
                         <expressão>
                       | <expressão>
<comando result>::= RESULT <expressão>
<chamada>::= <identificador> (<lista identificadores>)
           | <identificador>
<comando de E|S>::= READ (<lista formato>)
                  | WRITE (<lista formato>)
<lista formato>::= <lista formato>; <formato>
                   <formato>
<formato>::= <formato livre>
           | <formato não livre>
             <variavel E|S>
```