A tradução como método de programação (*)

por

José Augusto Legatheaux Martins

UNL - 9/81 Junho - 1981

(*) Lição submetida ao exame de habilitação à função de Assistente.

ÍNDICE

1. INTRODUÇÃO

1.1. O tema da lição

Os últimos anos têm assistido a um grande incremento do interesse pelos métodos que permitem um aumento da qualidade da programação. Tal incremento tem origem na chamada "crise do software", frase na qual se sintetiza toda a incapacidade em lidar com a crescente complexidade dos problemas informáticos que os métodos tradicionais vêm revelando.

Assim, todas as metodologias, total ou parcialmente formalizadas, das quais se possam inferir métodos de analisar e descrever problemas e métodos de estruturar programas e algoritmos para sua solução assumem grande importância.

Um destes métodos é o método da análise por refinamentos sucessivos (1) que consiste na transformação sucessiva de um enunciado num primeiro esboço de programa, que é então transformado num programa real, por uma sucessão de transformações que para ele convergem.

Um outro método que cada vez mais se afirma é o método da "programação pela sintaxe" ou da "tradução como método de programação"(2).

Tal método consiste em conceber um programa como uma aplicação do espaço de entradas no espaço de saídas, aplicação essa que pode ser assimilada a uma tradução da

---

(1) "Step wise refinement" na literatura de lingua inglesa.

(2) "Syntax directed translation", "Syntax directed programming" na literatura de lingua inglesa.

linguagem (1) que descreve o espaço das entradas na linguagem que descreve o espaço das saídas.

Também como no método dos refinamentos sucessivos, a tradução pode ser decomposta num número k ($k \geq 1$) de etapas intermédias, sendo cada "tradutor intermédio" uma componente bem definida do programa, cuja estrutura básica o método também permite determinar.

Assim, o programa é concebido como um analisador sintáctico da Gramática que descreve todas as suas possíveis entradas. A esse analisador são adicionadas "funções semânticas" que traduzem a entrada nas saídas adequadas. A estrutura do programa é a estrutura do analisador sintáctico. A linguagem de saída descreve a resposta do programa ou uma linguagem intermédia que um novo tradutor irá traduzir na resposta, e assim sucessivamente.

O método foi desenvolvido e sistematizado pelos "escritores de compiladores e interpretadores". O primeiro que sugeriu a sua aplicação de forma sistemática, fora do contexto da compilação, foi Jackson /1/.

1.2. Justificações da importância do método

Ainda que à primeira vista este método de programação só possa ser aplicado a categorias de problemas semelhantes àqueles que se encontram na compilação, tal ideia não corresponde à verdade e vários aspectos justificam a sua importância:

---

(1) Aqui linguagem é tomada segundo a definição formal, isto é, conjunto de palavras sobre um determinado alfabeto.

a) O método da programação pela sintaxe aplica-se com vantagem a qualquer problema cujos dados sejam estruturalmente complexos e sugiram a utilização de métodos sintácticos para a sua resolução. Tal é o caso de:

   - Tratamento de textos.
   - Programação interactiva em geral.
   - Manipulação de ficheiros de dados estruturalmente complicados.
   - Tratamento de mensagens em teleprocessamento.
   - Aplicações interactivas em tempo-real.
   - Manipulação de programas.
   - Interpretação de comandos
   - Etc., etc.

b) O método é justificado formalmente e corresponde a uma aplicação intensiva dos rudimentos da teoria das Linguagens Formais e Autómatos.

c) Mesmo sem aplicação sistemática daquela teoria, a programação pela sintaxe fornece indicações metodológicas importantes sobre a forma de encarar certos problemas.

d) A programação pela sintaxe ajuda à clarificação e especificação do problema.

e) Quando está disponível um "Parser Generator" (1) a sintaxe é utilizada como uma "linguagem de programação de muito alto nível" /2/, /3/.

1.3. Enquadramento da lição no contexto da licenciatura

A compreensão do método, quando se ultrapassa a simples utilização dos diagramas de transição, exige um conhe-

---

(1) Programa gerador de analisadores sintácticos

cimento mínimo da teoria das Linguagens Formais e dos Autó matos, assim como um conhecimento sucinto da implementação de um reconhecedor ou analisador sintáctico segundo o méto do LL(1), quando, como é o nosso caso, se suporta o método em linguagens com as propriedades LL(1).

Por outro lado, pressupõe-se que o programador tem conhecimentos de programação que ultrapassam o simples qua dro introdutório, nomeadamente, exige-se o conhecimento da recursividade.

Assim , este tipo de exigências leva a que o ensi- no do método da programação pela sintaxe deva ser inserido na parte final de uma cadeira de Linguagens Formais e Autó matos, ou na mesma fase do curso, mas enquadrado nas cadei ras de Técnicas de Programação.

A presente lição pressupõe que os alunos satisfa- zem os requisitos anteriores.

1.4. Objectivos da lição

O objectivo central da lição é ensinar o método da programação pela sintaxe. Sendo o aluno levado a compreen- der sucintamente como o método se aplica sucessivamente:

a) À fase de análise do problema.
b) À partição do problema em sub-problemas.
c) À obtenção de uma solução final.

Partindo do pressuposto de que o aluno tem conheci mentos mínimos da teoria das Linguagens Formais e dos Autó matos, assim como da forma como se implementam reconhecedo res destas linguagens, pretende-se levá-lo a compreender a ligação daquela teoria com a prática da resolução de pro- blemas correntes em Informática.

Como objectivos secundários pretende-se pôr em evi dência que:

a) Cada problema tem o seu "lado sintáctico" e o seu "la-

do semântico". Aprender a reconhecer e separar cada um deles. Clarificar os conceitos de sintaxe e de semântica.

b) Que a sintaxe concretizada em termos de gramáticas, expressões regulares ou gramáticas de expressões regulares é uma notação condensada e muito legível para exprimir o "lado sintáctico" de um problema.

c) Que uma especificação sintáctica pode ser imediatamente interpretada como um programa. O contrário também é válido em certas circunstâncias.

d) Que a especificação sintáctica é o veículo que suporta uma descrição semântica de uma forma mais natural.

e) Que assim como conceptualmente um problema pode por transformações sucessivas ser particionado em sub-problemas, também a aplicação do espaço das entradas no espaço de saídas pode ser concebida como uma composição de funções.

f) Que a análise de um problema segundo um método formalmente justificado conduz a soluções "em princípio" correctas, ou susceptiveis de se provarem certas com maior facilidade. Sendo nestas circunstâncias o ênfase central da programação passado para a fase de análise e concepção e não para a fase de implementação e teste.

g) Que a utilização do método da programação pela sintaxe tem a virtude de obrigar à explicitação de todas as questões escondidas, ou passadas em claro, num enunciado implícito de um problema.

h) Que os tradutores são programas criados no dia a dia das aplicações e não sómente pelas pessoas que desenham o "Software de sistema" e que portanto o conhecimento dos métodos com que aqueles programas são construídos é determinante em muitos ramos da informática aplicada.

O Conjunto da lição é proferida durante várias aulas. A uma primeira exposição teórica, devem seguir-se 1 ou 2 aulas práticas em que o docente procura resolvêr, em conjunto com os alunos, os problemas propostos.

Para que todo o carácter sistemático do método seja apreendido, é necessário complementar a lição com projectos individuais ou de grupo, para que as ideias amadureçam no aluno.

A experiência mostrou o carácter imprescindível, e também o sucesso, deste procedimento.

Para terminar esta introdução resta referir que as notações utilizadas são as por nós introduzidas em /3/.

2. CONCRETIZAÇÃO DOS OBJECTIVOS

2.1. Apresentação do método

Apresenta-se de seguida o encadeamento de noções e deias que são transmitidas ao aluno no sentido de o levar a apreender o método. Estas encontram-se definidas rigorosamente no ANEXO 1.

Os enfases centrais com que o método é ensinado são apresentados pela ordem com que são transmitidos ao aluno

a) Sintaxe e Semântica de um problema.

Muitos problemas podem ser especificados em termos semelhantes à aproximação com que se descreve uma linqua gem de programação, isto é, eles têm uma sintaxe e uma semântica.

Vários exemplos são apresentados nesse sentido.

Um editor interactivo tem como sintaxe a sua interface com o utilizador. É fácil de definir uma linguagem que "gera" todas as sessões de trabalho do utilizador sentado ao terminal (ver o respectivo exemplo). A semântica do editor é a descrição do significado do efeito dos comandos sobre o ficheiro em edição.

Uma linguagem de acesso a uma base de dados é um exemplo semelhante.
Num ficheiro contendo "actualizações" de um ficheiro de empregados a sintaxe é a descricão do ficheiro de "update", a semântica, o significado do efeito de cada registo sobre o ficheiro em actualização, /2/.

Vários outros exemplos põem em evidência esta ideia.
A linguagem de comunicação com um sistema de operação por exemplo.
Trata-se de levar o aluno a ir compreendendo a distinção entre sintaxe e semântica, essencialmente através de exemplos, dado que uma definição formal destes 2 as

pectos é impossível de fornecer neste quadro.

b) Sintaxe e semântica são 2 realidades distintas e como tal devem ser encaradas e modelizadas.

O exemplo de sistema de operação é muito adequado a pôr em evidência esta realidade.
A aproximação básica desta ideia central pode ser feita da-seguinte forma:
A forma concreta como os comandos para o sistema de operação são escritos é a sintaxe do problema. Exemplo

DELETE DEMO.ABS

Trata-se de uma ordem, num certo sistema de operação, para que o ficheiro de nome DEMO.ABS seja destruído e o espaço que ocupava em disco recuperado para a lista de blocos livres.
Esta última descrição é a semântica daquele comando. A sintaxe da interface do sistema de operação com o utilizador é a forma de escrever todas as sequências de ordens possíveis. A semântica é a descrição do efeito dessas ordens.
Num sistema de operação um módulo interpreta a "sintaxe", é o módulo de diálogo ("Comand Line Interpreter"), que após o reconhecimento sintáctico do comando desencadeia as operações semânticas correspondentes a uma da da sintaxe.
A semântica das operações fornecidas por um sistema de operação está encerrada no seu "Kernel"! Sintaxe e semântica são realidades diferentes e como tal devem ser encaradas.

c) O conceito de Tradutor e o conceito de Interpretador

O exemplo do sistema de operação pode continuar, devido à sua riqueza e familiaridade, a ilustrar estas noções.
Assim o "Comand line interpreter" não é mais que um tradutor da linguagem de comandos utilizada na consola, na linguagem definida pelo "Kernel" do sistema de operação.

Neste caso trata-se de um caso particular de tradutor, um interpretador, dado que o mesmo desencadeia imediatamente a execução da máquina abstracta capaz de executar as operações especificadas pelo programa traduzido.

d) Os métodos formais de descrição sintáctica clarificam a especificação desta categoria de problemas

Neste ponto pode ser pedido à classe que desenhe a gramática de uma linguagem que gere todas as sessões de trabalho possíveis com um conjunto restrito e bem conhecido de comandos para o sistema de operação.
Este exercício permite então ao docente pôr em evidência como esses métodos formais são adequados à descrição sintáctica de sistemas reais e tendem cada vêz mais a ser utilizados pelos próprios fabricantes.

Se a gramática for bem desenhada, e o docente deve orientar a classe nesse sentido, a cada comando corresponde uma produção. É então o momento de explicar como a sintaxe facilita a descrição semântica pois destaca cada categoria sintáctica (1 produção ou um conjunto de produções) com um significado semântico bem definido.

e) Conceito de "avaliação de um programa"

O conceito de "avaliação de uma função" deve então ser estendido ao conceito de tradutor.
Um calculador avalia uma expressão que se lhe apresente. Um calculador é um tradutor que aplica o espaço definido por uma linguagem de expressões particulares no seu resultado, isto é, no espaço das imagens (em geral o conjunto dos reais). Um tradutor avalia um programa traduzindo-o. Aqui, programa é tomado no seu sentido lato, isto é, qualquer sequência válida de símbolos sobre um dado alfabeto.

A avaliação que um compilador faz do programa fonte é a sua tradução em código objecto. A avaliação que o "Command Line interpreter" faz da linha comunicada pelo utilizador é o conjunto de acções semânticas correspondentes executadas pelo "Kernel" de sistema de operação.

f) Relações entre a avaliação e a sintaxe

Neste ponto deve-se comunicar a ideia de que a maneira adequada de implementar um programa que "avalia" uma sequência de símbolos que se lhe apresente à entrada é desenhar um analisador sintáctico dessa sequência e acrescentar-lhe nos pontos que marcam no processo de reconhecimento da entrada o reconhecimento de uma fase correcta, a chamada da função semântica que calcula aquela parte do conjunto imagem.

Esta é a essência do método da programação pela sintaxe que deve sêr bem explorada nos problemas que completam a parte teórica da lição.

Neste ponto e em função do tempo disponível o docente pode apresentar alguns exemplos clássicos de avaliação de expressões com parentesis que são suficientemente simples para serem rapidamente explicados e suficientemente complexos para serem quase impossíveis de tratar por outro método que não o método da programação pela sintaxe.

Chegados a este ponto pode-se passar imediatamente à discussão dos problemas, ou optar por um conjunto de considerações que, naquela hipótese, são feitas ao longo da solução dos mesmos.

Em qualquer das hipóteses no final da apresentação dos problemas essas considerações devem ser retomadas e rediscutidas com os alunos.

São elas as seguintes:

a) Relações entre a programação pela sintaxe e a análise por refinamentos sucessivos.

Se a avaliação da entrada é difícil de conceber de uma só vez (em 1 só passo podem-se conceber uma sucessão de avaliações que convergem para o resultado pretendido.

Trata-se de conceber a função de avaliação como uma composição de funções, ou de conceber um tradutor como uma sequência de tradutores em vários passos, como se diz em compilação.

Aqui, há que chamar a atenção para a semelhança da aproximação aos problemas feita pelo método da programação por refinamentos sucessivos. Também como nesse método, a solução final pode ou não reflectir a solução

em fases. (Expansãode procedimentos inúteis no método da programação por refinamentos sucessivos, fusão de vários tradutores num só, no nosso caso).

Quer o exemplo do telecarregador quer o exemplo da Árvore de declarações permitirão ilustrar concretamente este aspecto.

Para terminar deve-se também realçar que um problema pode ser analisado de forma descendente ou de forma ascendente ("Top-down" ou "Bottom-up"), isto é, caminhando do espaço de partida para o espaço de chegada da função de avaliação ou inversamente.

b) Em súmula, deve-se ajudar o aluno a sintetizar os princípios fundamentais do método:

- Definir a sintaxe das sequências de símbolos válidos à entrada do programa.
- Defenir a semântica dessas sequências em termos de operadores de uma estrutura de dados abstracta, ou de um tipo de dados abstracto ou máquina abstracta, se nos quizermos situar num quadro mais formal.
- Conceber o programa como um reconhecedor sintáctico que nos pontos adequados da árvore de derivação da entrada, isto é, quando uma sub-frase é reconhecida, desencadeia os operadores semânticos associados à mesma.
- Este processo pode ser concebido em 1 ou várias fases e por ordem ascendente ou descendente.
- As notações sintácticas suportam com a sua clareza e o seu rigor formal o essencial do processo de análise.
- Dado que a construção de um reconhecedor sintáctico de uma linguagem com as propriedades LL(1) é um processo sistemático e que pode ser automatizado, o programa obtido está certo se a gramática e as funções semânticas estiverem certas.

2.2. Apresentação e discussão dos exemplos

Em anexo são discutidos 3 problemas cuja apresentação resolução e discussão, suportam a continuação da lição. Os exemplos foram por nós propostos e resolvidos e são retirados de /3/.

Cada problema encontra-se enunciado, resolvido e discutido com suficiente clareza. Sendo a forma como devem ser discutidos com os alunos bem evidenciada.

De seguida apresentam-se considerações individualizadas sobre cada um deles, quer no sentido de clarificar a intenção pedagógica que preside à sua apresentação, quer no sentido de os relacionar com a primeira parte desta aula.

2.2.1. Exemplo do telecarregador

A apresentação deste problema é dividida em 2 partes. Na primeira o enfase é para o seu aspecto sintáctico. Procura-se fixar a atenção do aluno nas possibilidades das gramáticas e expressões regulares em descrever sequências de símbolos quaisquer ao mesmo tempo que se lhes confere uma estrutura. Como é do conhecimento geral essa estrutura pode ser formalizada em termos da árvore de derivação de uma sequência válida na gramática escolhida.

Ainda que nesta fase apenas se procure fixar o aspecto meramente sintáctico do problema, o conceito de de legibilidade de uma gramática é introduzido. Assim, a estruturação em 2 níveis da sintaxe, assim como uma escolha criteriosa dos identificadores dos não terminais e dos terminais das gramáticas, sugerindo o seu significado semântico, é explorado para efeitos de clarificação do conceito.

Este aspecto da estruturação sintáctica e a sua relação com a estruturação do programa é também posto em evidência, chamando-se desde logo à atenção do aluno para tal facto. Estabele-se assim um paralelo nítido entre a aproxima-

ção por via da programação pela sintaxe e por via do método da análise por refinamentos sucessivos.

Na segunda parte do problema é então tratado o seu aspecto semântico e a relação directa da sintaxe com os algoritmos e da transformação de gramáticas com a transformação de algoritmos.

A semântica de um telecarregador tem que necessáriamente ser descrita em termos das instruções "STORE" e "JUMP" de um computador. O programa final é portanto um tradutor que avalia a sequência de simbolos presentes na sua entrada e os transforma num programa em memória cuja execução lança no fim.

A solução apresentada tem um carácter didáctico e por isso resolve o problema em 2 passos. Na prática tal não seria necessário e uma nota final mostra a solução que qualquer programador de sistema adoptaria.

Para terminarmos a apresentação resta chamar a atenção que os programas de controle dos protocolos de comunicação em redes de computadores podem e devem ser concebidos com êxito segundo a mesma aproximação. Na verdade mais não fazemos que formalizar uma técnica presente desde há longa data na programação, a saber, a utilização de diagramas de transição como "flow-chart's" de programas.

2.2.2 Exemplo do editor interactivo

No exemplo seguinte é feita a análise de um editor interactivo. Trata-se de levar o aluno até uma nova zona de aplicações, a do tratamento de texto e da programação interactiva.

Neste problema o enfase é para as ligações entre a sintaxe e o conceito de "conforto de utilização" de um programa.

Programando pela sintaxe é fácil de planear quais as sequências que devem claramente ser assinaladas com erros e qual o grau de flexibilidade com que se permite ao utilizador especificar as suas intenções sem que no entanto se deixe passar em claro sequências ambíguas.

Esta discussão está intimamente relacionada com a política de tratamento de erros implementada no reconhecedor sintáctico e da sua profunda relação com considerações de carácter ergonómico e de eficácia dos programas.

O âmbito do problema permite chamar a atenção para a estrutura, afinal muito semelhante, da grande maioria dos programas interactivos.

Finalmente o exemplo permite dar um exemplo completo (e afinal surpreendentemente claro) da semântica associada a uma linguagem de edição de textos.

2.2.3 - Exemplo da árvore de declarações de procedimentos e funções de um programa Pascal

O último exemplo apresenta facetas novas e com significados dignos de realce.

Assim, trata-se de um exemplo cuja sintaxe é descrita por uma linguagem não regular e que portanto não tem uma solução suportada num automato finito. Ainda que uma solução iteractiva para este problema seja possível, a solução recursiva subjacente à gramática exposta é a mais natural e a mais fácil de mostrar como certa.

O outro aspecto interessante que o exemplo permite realçar, é o facto de a função de avaliação da entrada ter sido desenhada por aproximações, isto é, com diversos tradutores em sequência, e do fim para o princípio, isto - é, por ordem inversa ao processo do seu cálculo.

Põe-se assim em evidência a versatibilidade do método.

Reflectindo na razão de ser desta inversão, tal acaba por surgir como natural e a sua explicação é a seguinte:

A árvore das declarações de um programa Pascal está diluida à entrada num sem número de pormenores. Por outro lado, a forma especial como se pretendia que o programa a apresentasse à saída, torna aparentemente difícil de conceber a função de uma só vez.

Ora a representação de uma árvore tal como o problema exige é regida pelas seguintes asserções:

a) Cada nó aparece isolado numa linha.
b) Os nós aparecem pela ordem com que são "visitados" num percurso prefixado da árvore.
c) A distância da margem esquerda à posição no papel em que o valor do nó (o seu identificador) é escrito, é o produto de um factor constante pela profundidade do nó na árvore.

Sendo o algoritmo de cálculo da profundidade do nó

um algoritmo bem conhecido é natural que se transforme o problema inicial em 2 sub-problemas:

a) Traduzir o programa Pascal numa linearização do percurso pre-fixado da sua árvore de declarações.
b) Traduzir aquela representação linear no aspecto gráfico pretendido.

É esta a razão de ser daquela análise.

Finalmente alguns dos tradutores são incorporados noutros pois o seu papel só se revelou útil durante a fase de análise.

O problema deveria terminar por uma discussão dos diferentes métodos para ligar os vários tradutores:

a) Relação como sub-rotinas uns dos outros.
b) Por meio de ficheiros intermédios.
c) Como Co-rotinas.
d) Processo concorrentes comunicantes.

3. BIBLIOGRAFIA

/1/ - Griffiths, Cunin.
Programmation dirigée par les données (CRIN - Nancy). 1980.
'Gramáticas e automatos são usados para desenhar sistemáticamente programas de gestão de stocks'.

/2/ - D. Coleman.
the Systematic Design of file-processing Programs. Software - Practice and Experience vol. 7 pág. 371.1977.

/3/ - J. Legatheaux Martins, Luís Monteiro.
Linguagens Formais e Automatos.
Universidade Nova de Lisboa 1981 (CIUNL).
(Notas de curso em vias de publicação).

/4/ - A. I. Wasserman, S. K. Stinson
A Specification method for interactive Information Systems - in
Procedings of the Conference on Specifications of Reliable Software
IEEE Nº 79 c 1 401-9c .1979

ANEXO 1

Definições e exemplos que suportam a apresentação do método

O presente anexo contem as definições e alguns dos exemplos que servem de suporte à exposição do método da "programação pela sintaxe" ou por outras palavras, da "tradução enquanto método de programação".

a) Definição de sintaxe.

Formalmente uma linguagem é um certo conjunto de palavras sobre o conjunto alfabeto. Esse conjunto de palavras pode ser descrito em extensão ou através de um conjunto de regras que o permitem descrever por compreensão.

Os formalismos mais utilizados para descrever linguagens em compreensão são:

- Gramáticas (mecanismo gerador).
- Expressões regulares (mecanismo "denotacional") (1).
- Automatos (mecanismo reconhecedor).
- Gramáticas de expressões regulares (mecanismo gerador e "denotacional").

Sob este ponto vista não se associa nenhum "significado" às frases da linguagem. Diz-se então que se está num ponto de vista meramente sintáctico ou simplesmente perante a sintaxe da linguagem.

Informalmente, a sintaxe de uma linguagem é o conjunto de regras que permitem descrever frases nessa linguagem independentemente do seu significado.

b) Definição de semântica.

A Semântica é o significado das frases de uma linguagem. A descrição formal da semântica não cabe no âmbito desta lição. Adoptaremos a noção informal de Semântica operacional.

A Semântica de uma frase é o efeito da mesmo numa máquina abstracta capaz de a compreender.

Por exemplo, a semântica da linguagem máquina de um computador é descrita operacionalmente como uma função de

mudança de estado no seu espaço de valores (conjunto dos registos + memória).

c) Sintaxe e Semântica de um problema.

Noção informal que se refere à sintaxe e semântica de uma linguagem que está subjacente ao problema.

d) Tradutor.

Tradutor é um programa que aplica um programa da linguagem fonte num programa "equivalente" da linguagem objecto.

Dada uma linguagem fonte, cuja semântica é explicada sobre uma máquina abstracta A, podemos dizer que a máquina abstracta B, que explica a semântica da linguagem objecto, mais não é que uma simulação da máquina A. Assim, os programas dizem-se equivalentes se a execução do programa objecto sobre B simula a execução do correspondente programa fonte sobre A.

Um interpretador é um tradutor especial que simultâneamente desencadeia as acções semânticas na máquina B.

e) Descrição da semântica suportada na sintaxe.

A sintaxe de uma linguagem é, quando descrita em termos de uma gramática, uma hierarquia de construções sintácticas (os seus não terminais). A cada uma dessas construções sintácticas associa-se em geral um significado semântico.

A gramática diz-se legível quando também facilita aquela tradução.

f) Exemplo de descrição sintáctica de um sub-conjunto de comandos num sistema de operação.

Gramática em BNF:

<Sessão de trabalho >::= <Sequência de comandos>

---

(1)- "Denotacional" no sentido em que se diz: A linguagem denotada pela seguinte expressão regular...

<Sequência de comandos>::= <Comando> <Sequência de comandos>|λ

<Comando>::= <COMPILAR>|<CARREGAR>|
<SUPRIMIR>|<EDITAR>

<COMPILAR>::= "PASCAL" <ficheiro> "CR"|"FORTRAN"<ficheiro> "CR"

<IMPRIMIR>::= "PRINT" <abreviatura> "CR"

<CARREGAR>::= "LOAD" <lista de ficheiros> "CR"

<SUPRIMIR>::= "DELETE" <abreviatura>"CR"

<EDITAR>::= "EDIT" <ficheiro> "CR"

<ficheiro>::= <nome> <extensão>

<extensão ::= "." <nome>|λ

<lista de ficheiros>::= <ficheiro> <lista de ficheiros>|λ

<abreviatura>::="*.*"|"*"<extensão>|<nome>".*"

<nome>::= <letra> <resto>

<resto>::= <letra>|<digito>|λ

<letra>::= "A"|...|"Z"

<digito>::= "0"| ...|"9"

Nota: "CR" é a abreviatura de "carriage return" (tecla de).

g) Conceito de avaliação.

Sob o ponto de vista da tradução a avaliação de um programa é o resultado da sua tradução. Sob o ponto de vista da interpretação a avaliação de um programa é o resultado da execução do programa traduzido.

Um compilador é um interpretador de uma máquina de tradução do código fonte em código objecto.

h) Analisador sintáctico.

É um programa específico de uma dada linguagem e que tendo por entrada um programa dessa linguagem tem 2 respostas possíveis: Ou assinala pelo menos um erro sintáctico e diz que o programa não pertence à linguagem

que é capaz de reconhecer, ou diz que o programa pertence à linguagem que reconhece e neste caso recompõe a árvore de derivação do programa na Gramática da linguagem que reconhece.

i) Programação pela sintaxe.

Método que consiste em conceber um programa como um tradutor. O programa é construído incorporando funções semânticas no analisador sintáctico da linguagem fonte a traduzir. As funções semânticas realizam a tradução segundo o princípio da equivalência semântica e têm como resultado a semântica associada à frase da linguagem no entretanto reconhecida. Por este motivo são inseridas nos pontos em que o analisador sintáctico termina a construção da árvore de derivação daquela frase.

j) Construção de um tradutor por passos.

Método que consiste em decompor a função de avaliação de um programa na composição de um conjunto de funções de avaliação intermédias. Cada tradutor intermédio é concebido segundo o mesmo método que o tradutor global.

A forma de conceber as fases intermédias é indiferente. Do princípio para o fim, do fim para o princípio, ou de forma mista.

k) Exemplos sobre a avaliação de expressões aritméticas simples.

Dada a gramática simplificada de expressões aritméticas:

```
E = T {'+'T}.
T = F {'*'F}.
F = "número" + "(" E ")".
```

Onde E significa Expressão, T termo e F factor, com a semântica convencional das expressões aritméticas.

Temos:

K.1) Avaliação da expressão por um único tradutor.

E = /*acc:=(T)*/{'+'/*acc:=acc+(T)*/} /*devolve acc*/.
T = /*acc:=(F)*/{'*'/*acc:=acc*(F)*/} /*devolve acc*/.
F = /*acc:='número')*/ + '(' /*acc:=(E)*/ ')' /*devolve acc*/.
onde (x) representa valor devolvido pela chamada de x.

Programa:

```
function E: integer;
    begin
        acc:= T;
        while terminal='+' do
                        begin
                            ler (terminal);
                            acc:=acc+T
                        end
    end;
function T:integer;
    begin
        acc:=F;
        while terminal ='*' do
            begin
                ler (terminal);
                acc:=acc*F
            end
    end;
function F:integer;
    begin
      if terminal='número'then
            begin
                F:=valor('número');
                ler (terminal)
            end
      else
            begin
                ler(terminal);
                F:=E;
```

```
                    ler (terminal)
                end
        end;
```

E devolve o valor da expressão que se lhe apresente à entrada.

Utilizam-se as nomenclaturas introduzidas em /3/.

K.2) Avaliação da expressão por 2 passos.

Para a mesma gramática. Começa-se por apresentar o tradutor que traduz a expressão em post fixado.

```
E = T {'+'T/*escrever('+')*/}.
T = F {'*'F/*escrever('*')*/}.
F = 'número'/*escrever('número')*/'('E')'.
```

Da qual se deduz directamente o algoritmo.

Em seguida apresenta-se uma gramática possível para expressões post fixadas que através de funções semânticas sobre uma pilha, avalia a expressão.

```
Epf =  Operador + Operando  .
Operando = "número" /*empilhar ("número")*/.
Operador  = "+" /* op1:= topo; desempilhar;
                   op2:= topo; desempilhar;
                   empilhar (op1+op2);*/ +
                   '*' /* op1:=topo; desempilhar;
                   op2:=topo; desempilhar;
                   empilhar (op1*op2);*/.
```

No fim o resultado está no topo da pilha.

ANEXO 2

Problema do telecarregador

Modelo de um Tele carregador

Um microcomputador de 16 bits e 64 K de memória está ligado a um computador, por uma linha pela qual o micro recebe "bytes" enviados pelo outro computador.

Um byte especial ∝, assinala que o byte seguinte tem um código de operação. O envio do próprio byte ∝ é transformado no envio de ∝∝.

a) envio de 1 bloco de código a carregar. Para isso é necessário enviar o código de operação ∝A - início de bloco - em seguida 2 bytes com o endereço de início do carregamento - seguidos dos bytes a carregar até um código de operação ∝F - fim de bloco de carregamento.(1)

b) envio de 1 bloco com indicação de endereço de início da execução que tem a forma:

   Código de operação: ∝β, 2 bytes com o endereço de início da execução seguidos de ∝F.

O computador pode enviar vários blocos para carregar, terminados por um único bloco de lançamento da execução.

Escreva a gramática que gera todas as sequências possíveis da comunicação no sentido computador -> micro.

Classifique a gramática e linguagem obtidas.

Assegure que a gramática é "limpa".

Este é um exemplo típico que ganha em clareza se for resolvido em 2 etapas. Uma primeira gramática gera todas as sequências de bytes transmissíveis tratando apenas da sua micro estrutura:

<atomo-transmitido>::= <código> | <outros>
<código >::= <abrir-carregar>|<abrir-executar>|<fim>
<abrir-carregar>::=∝A
<abrir-executar>::=∝B

---

(1) - Os bytes a carregar são sempre em número par.

<fim> ::= ∝F
<outros>::= <atomo∝>|<simples>
<atomo∝>::= ∝∝
<simples>::= qualquer outro byte diferente de ∝ .

Repare-se que ∝ , A, B, F são designações abstractas para um determinado padrão binário de 8 bits.

Repare-se também, que a seguinte transmissão, do ponto de vista dos atomos transmitidos, não é proibida pela gramática, apesar de não ter significado enquanto transmissão:

$$A\ B\ X_1\ X_2\ X_3\ X_4\ \propto F\ X_5\ F\ \ldots$$

No entanto do mesmo ponto de vista

$$X_2\ \propto K\ \propto A\ \propto B\ \propto F\ X_1\ X_2 \propto F\ \ldots$$

é ilegal pois ∝K é uma sequência de bytes que não pertence à linguagem gerada pela gramática. De uma forma sistemática temos:

<axioma> - <atomo transmitido>
conjunto dos treminais = padrão de 8 bits
" de não treminais = <atomo-transmitido>, <códigos>, <simples>, <atomo ∝>,<outros>, <abrir-carregar>,<abrir-executar>,<fim>

Todas as produções são das formas:

$$A \rightarrow XB \mid X \quad \text{com} \quad X \in T^* \quad \text{e} \quad A, B \in N$$

Logo a gramática é linear direita.

Quanto à sua "limpeza" é evidente que o é, visto que gera a mesma linguagem que:

<atomo-transmitido> ::= ∝A | ∝B | ∝F | ∝∝ | $\overline{qq}$ outro ≠ ∝

que se obtem substituindo alguns não terminais na parte direita das produções sistemáticamente até só termos terminais.

<atomo-transmitido>::= <códigos>|<atomo >|<simples>

⋮

<atomo-transmitido>::= ∝A | ∝B |∝F |<atomo∝>|<simples>

⋮

O método em geral não se pode aplicar (!) senão a gramáticas muito simples.

A gramática foi apresentada com aquele maior número de produções para que seja mais "legível".

Vamos então agora construir outra gramática cujo conjunto terminal está contido no conjunto dos não terminais da primeira e que vai dar a solução final ao problema.

Gramática que denota as transmissões possíveis:

<transmissão>::= <carregamento><lançamento execução>
<carregamento>:: = <segmento><carregamento>|λ
<segmento>::= abrir-carregar outros outros
<segmento-programa> fim
<segmento-programa>::= outros <segmento-programa>|λ
<lançamento-execução>::= abrir-execução outros outros
fim

A gramática que gera por exemplo a palavra

abrir-carregar outros outros outros outros ...
outros fim

abrir-execução outros outros fim

Se outros denotar um byte simples da outra gramática (∝ ou ≠) e se substituirmos estes terminais pelos terminais respectivos

da outra gramática obtemos a seguinte sequência de bytes

∝ A X X X X X ... X ∝ F ∝ B ∧ X ∝ F

que é uma palavra correspondente a uma transmissão correcta.

Esta 2ª gramática também denota uma linguagem regular e tem por:

axioma: <transmissão>

Não terminais: <carregamento>, <lançamento-execução> <segmento>,<segmento programa>

Terminais: abrir-carregar, outros, abrir-execução, fim

É fácil verificar que as gramáticas estão limpas

Finalmente um aspecto que importa desde já chamar a atenção, é o seguinte:

Ver-se-á adiante que sempre que se está em presença de uma gramática context-free limpa(1) se podem programar directamente algorítmos que reconhecem se as palavras pertencem à linguagem gerada pela gramática e que simultâneamente reconhecem a estrutura da derivação que conduziu à mesma. Em princípio a cada regra da gramática corresponderá um procedure que trata dessa produção. Nesses procedimentos pode-se inserir o código que realiza as acções "semânticas" correspondentes. Por exemplo, o procedure correspondente a<lançamento-execução>, além de reconhecer um bloco deste tipo, executaria no fim, a acção semântica Branch incondicional para o endereço denominado por <outros> <outros>.

É assim que a legibilidade de uma gramática está em profun-

---

(1)- Para o algoritmo ser simples e eficaz é também necessário garantir outras condições o que de uma forma geral se consegue fácilmente aplicando algumas transformações à gramática se isso for necessário.

da relação com a legibilidade do algoritmo, assim como a solução em "camadas" de gramáticas corresponde profundamente às soluções por "camadas" de software.

O método da programação descendente por refinamentos sucessivos, consiste em dar uma solução global a um problema, pressupondo grandes operações primitivas que de facto não existem na linguagem de programação que se usa. Na continuação do desenvolvimento do algoritmo essas operações vão sendo sucessivamente pormenorizadas em outras de mais baixo nível e assim sucessivamente. Um programa é tanto mais legível quanto a sua estrutura reflicta o traço do seu processo de desenvolvimento.

Na análise de um programa que reconhecesse o input vindo do computador começariamos por:

```
início
   enquanto houver segmentos para carregar fazer
                                          carregar;
   lançar a execução;
fim.
```

A certa altura do processo de desenvolvimento do programa teriamos necessidade de recorrer a uma operação de leitura de um código de input. É claro que poderiamos ler directamente o byte recebido mas é muito mais interessante dispor-se de um procedimento

```
get-next-atom (var B: Byte,var tipo: integer);
```

que resolvesse todos os pequenos aspectos que têm a ver com os problemas provocados pelo $\propto$ como prefixo. Ora este procedimento está em profunda relação com a primeira gramática pois o trabalho que ele realiza é exactamente o de reconhecer os atomos transmitidos no input e se ele fosse desenhado sistemáticamente ele teria procedimentos internos relacionados com as produções dessa gramática.

Neste caso concrecto a gramática não se poderia usar na forma em que está. A título meramente ilustrativo apresenta-se a gramática que poderia ser usada para a programação:

<atomo-transmitido>::= $\overline{\alpha\alpha}$ byte ≠ de α | α<resto>
<resto>::= A|B|F|α

Assim a partição em "camadas" de gramáticas está profundamente relacionada com a partição em "camadas" de software ou com a partição de um programa em sub-problemas. Resta responder à questão: porquê começar pela gramática apresentada em 1º lugar se ela é o refinamento final do problema ? A resposta é simples: O método dos refinamentos sucessivos não é para aplicar como um "dogma" ! Para certas categorias de problemas reconhece-se logo à partida que vai ser necessário um conjunto de procedimentos para tratar outro sub-problema que se identifica fácilmente. Impor que o desenho poderia começar por aí seria puro "academicismo".A programação exige sempre muita experiência e invenção!

Para terminar resta dizer que uma solução do problema em uma só gramática, pode ser fácilmente obtida introduzindo na 2ª gramática as produções da 1ª que descrevem os "terminais" outros, abrir-execução, abrir-carregamento e fim. O que sob o ponto de vista da programação corresponde a uma substituição de procedimentos pelas instruções que estes representam.

Problema do Telecarregador - algoritmos

A gramática de baixo nível define o conjunto dos terminais da gramática de alto nível. ESte conjunto é o sub-conjunto do conjunto dos não terminais da gramática de baixo nível:

{<abrir-carregar>,<abrir-executar>,<fim>,<outros>}

A gramática de baixo nível é:

<átomo>::=<outros>|<fim>|<abrir-executar>|<abrir-carregar>

<abrir-carregar>::= ∝A

<abrir-executar>::= ∝B

<fim>::= ∝F

<outros>::= ∝∝ |$\overline{qq}$ byte ≠ de ∝ .

O procedimento ler-terminal, da gramática de alto nível, é um automato que lendo bytes do input de baixo nível atinge um estado final sempre que reconhece um daqueles átomos.

Para obtermos esse automato transformamos a gramática até obtermos a expressão regular que denota a mesma lin-

guagem:

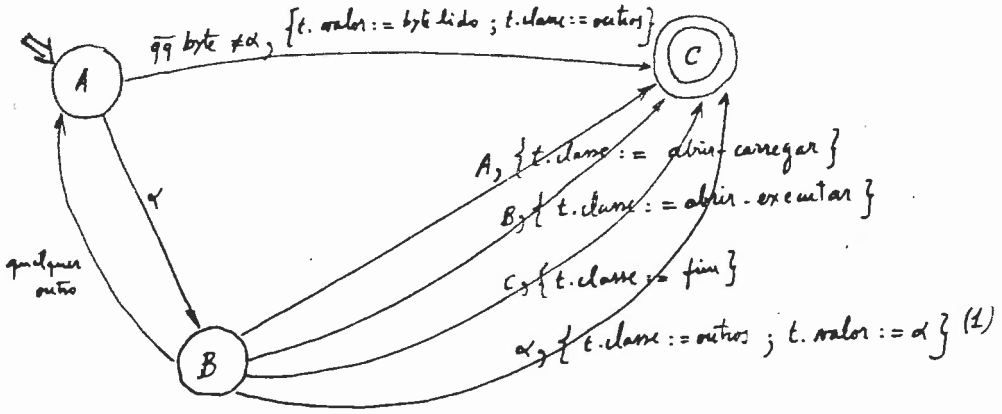
$$\propto (A + B + F + \text{qq}) + \overline{\text{qq}}\ \text{byte} \neq \propto$$

Assim a gramática de alto nível trabalhará sobre

```
type classe-terminal = (outros, abrir-carregar,
                        abrir-executar, fim);
     conjunto-terminal = Record
                           classe: classe-terminal;
                           valor: byte
                         end;
```

O procedimento ler-terminal dessa gramática é modeliza do pelo seguinto automato:

```
procedure ler-terminal (var t: conjunto-terminal)
```



(1) porque a semantica de 2 $\propto$'s seguidos é que o valor transmitido é $\propto$.

No exemplo introduzimos um tratamento de erros que consiste em desprezar o caracter errado.

Este automato tem um único estado final e seja qual for o estado em que está, existe um estado seguinte para qualquer que seja o byte no input. Em cada activação de ler-terminal o automato arranca no estado A e quando terminar em C colocou em t o terminal colectado do input.

```
procedure ler-terminal (var t: conjunto-terminal);
    const ∝ = ...;
          A = ...;
          B = ...;
          F = ...;
     type conjunto-estados = (stA, stB, stC);
      var tt: byte; s: conjunto-estados;
    begin
         s: = stA;
         ler (tt);
         while s <>stC do
            case s of

                StA: if tt<>∝ then begin
                                     t.valor:=tt;
                                     t.classe:=outros;
                                     s:=stC
                                   end
                               else begin
                                     ler (tt);
                                     s:=stB
                                   end;

                StB:
                    if tt = A then begin
                                     t.classe:=abrir-carregar
                                     ler (tt);
                                     S:=StC
                                     end
```

```
                else if tt = B then ...
                else if tt = C then ...
                else if tt = ∝ then ...
                else begin
                          ler (tt);
                          S:=sta
                     end;

          StC:;

        end (* case *)
      end (* ler-terminal *);
```

Analisemos agora o problema da gramática de alto nível.

Em primeiro lugar constatamos que não é uma gramática regular. No entanto se a transformarmos sucessivamente, quer por substituições, quer transformando a recursividade pela introdução do operador *, obtem-se a expressão regular:

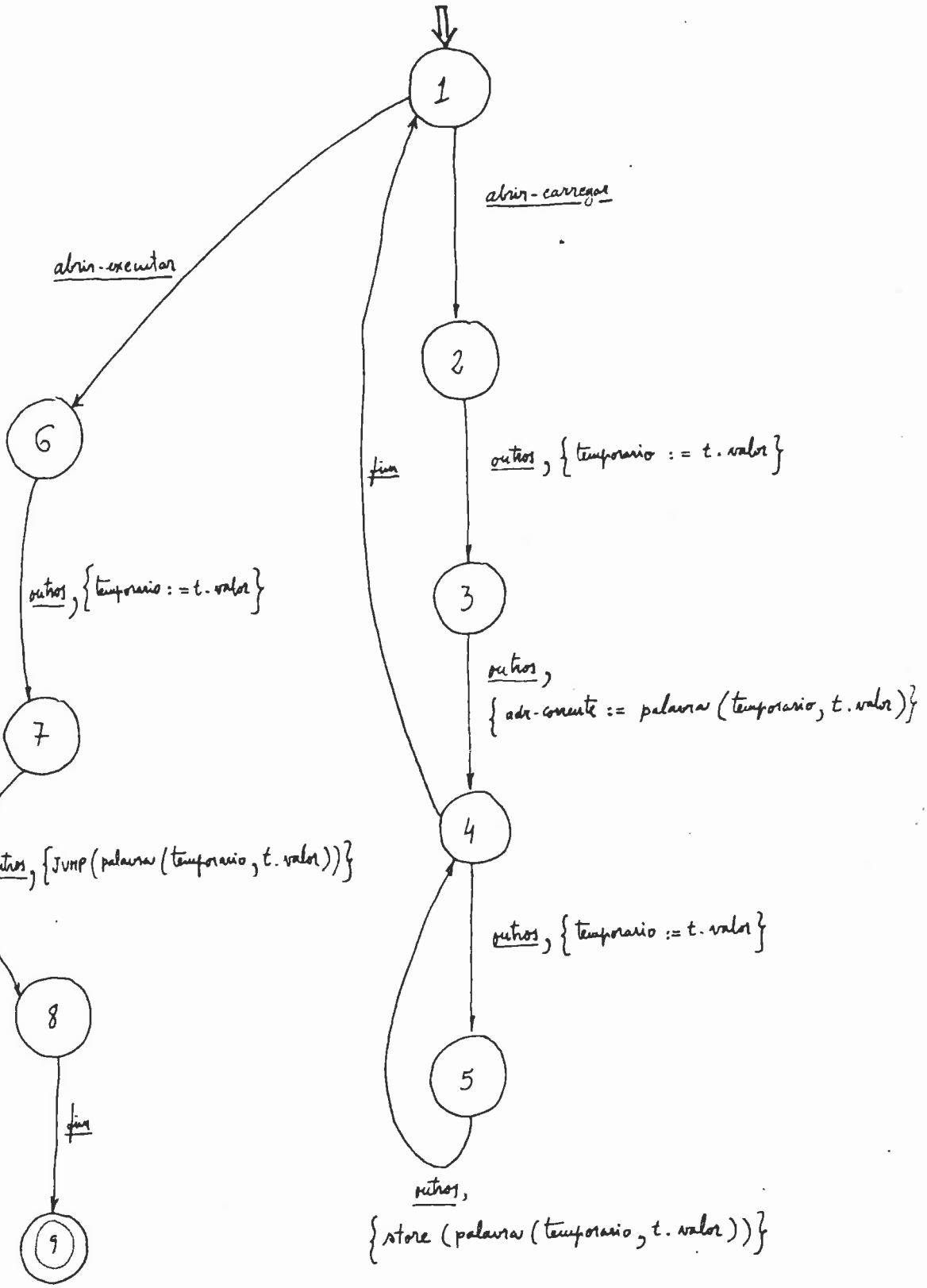
<transmissão>≡
(abrir-carregar outros outros (outros outros)*
fim)* (abrir-executar outros outros fim)

Logo estamos perante uma linguagem regular para a qual é possível determinar um automato finito determinista que a reconhece.

No que se segue vamos admitir que não há erros.

Para explicarmos sintéticamente a semântica do problema vamos introduzir as seguintes definições:

```
type word = ... (*palavra de memória*)
     byte = 0 ..  256;
var temporário: byte;
     adr-corrente: word;
     t: terminal;
function palavra (b1, b2: byte): word;
     begin
          palavra: = b2 "concatenado com" b1
     end;
procedure store (P: word);
     begin
          memory (adr-corrente): = p;
          adr-corrente: = adr-corrente + 1
     end;
procedure JUMP (adr: word);
     (* lança a execução em adr *)
```


1
abrir-carregar
abrir-executar
2
6
outros, {temporario := t.valor}
fim
outros, {temporario := t.valor}
3
outros,
{adr-corrente := palavra(temporario, t.valor)}
7
4
outros, {JUMP(palavra(temporario, t.valor))}
outros, {temporario := t.valor}
8
5
fim
outros,
{store(palavra(temporario, t.valor))}
9

Há laia de conclusão, podemos vêr como as gramáticas, expressões regulares e automatos, foram usados para analisar e obter um algoritmo, que é um modelo de uma solução para o problema colocado inicialmente.

Quais as questões a pôr em evidência:

a. As gramáticas ajudaram à clarificação da especificação do problema.

b. Ajudaram a conduzir a análise do mesmo.

c. Permitiram após manipulações adequadas das mesmas, obter algoritmos mais eficientes para implementar uma sua solução.

   Assim, a análise do algoritmo foi feita, de forma interposta, na transformação das gramáticas.

d. Os automatos poseram claramente em evidência os pressupostos que se tiveram que fazer, no que toca ao tratamento de erros.

NOTA COMPLEMENTAR:

A solução apresentada não é realista no que toca à utilização do procedimento ler-terminal. Um Tele-carregador destina-se geralmente a ser instalado numa máquina nua, sendo a sua função o carregamento do sistema de arranque (boodstrap loader). Todo o algoritmo desenhado é um modelo fácil de traduzir em código assembler, com excepção exactamente de ler-terminal, que exige uma possível "bufferização" da leitura de bytes, ou qualquer outro processo de bloquear o algoritmo até um byte chegar pela linha.

Uma outra solução seria fundir as 2 gramáticas e obter um único autómato. A iniciativa deveria então pertencer ao "handler" da linha de input, que, sempre que recebe um byte da linha, executa o case statement global do autómato.

Este por sua vez deixa de estar envolvido no while.

```
procedure  I/O trap (B: byte (*byte recebido*));
    begin
        case estado of
            1:
            2:
            3:
            .
            .
            .
            n: ...
               JUMP (palavra (temporário, B))
        end;
    end (* I/O trap *);
```

Todas as varáveis (estado, temporário, etc.) seriam globais.

ANEXO 3

Problema do editor interactivo

Um editor interactivo - exemplo reduzido

Especificação reduzida.

Um editor interactivo, após o lançamento da execução, lê para memória central um ficheiro de texto, passado em input.

Os seguintes comandos permitem modificar esse texto (Inicialmente a linha corrente é a linha imediatamente anterior à primeira do ficheiro):

a. Inserir uma linha antes da linha corrente:
   Formato: I sequência de caracteres (carriage return)
   A linha corrente não se altera.

b. Suprimir a linha corrente:
   Formato: S (CR)
   A linha corrente é a que se segue.

c. Saltar relativamente linhas:
Formato: ± inteiro B (CR)
O sinal + pode ser omitido.

d. Mostrar a linha corrente:
Formato: M (CR)

e. Fim da edição:
Formato: F (CR)

Uma expressão regular que denota todas as sessões de trabalho sintácticamente correctas é a seguinte:

("I" "C" * "CR" + "S" "CR" + ("+" + "-" +λ) "d" "d" *
"B" "CR" + "M" "CR")* "F" "CR"

em que os terminais aparecem entre "'S.
"C" significa qualquer caracter
"CR" carriage return
"d" digito

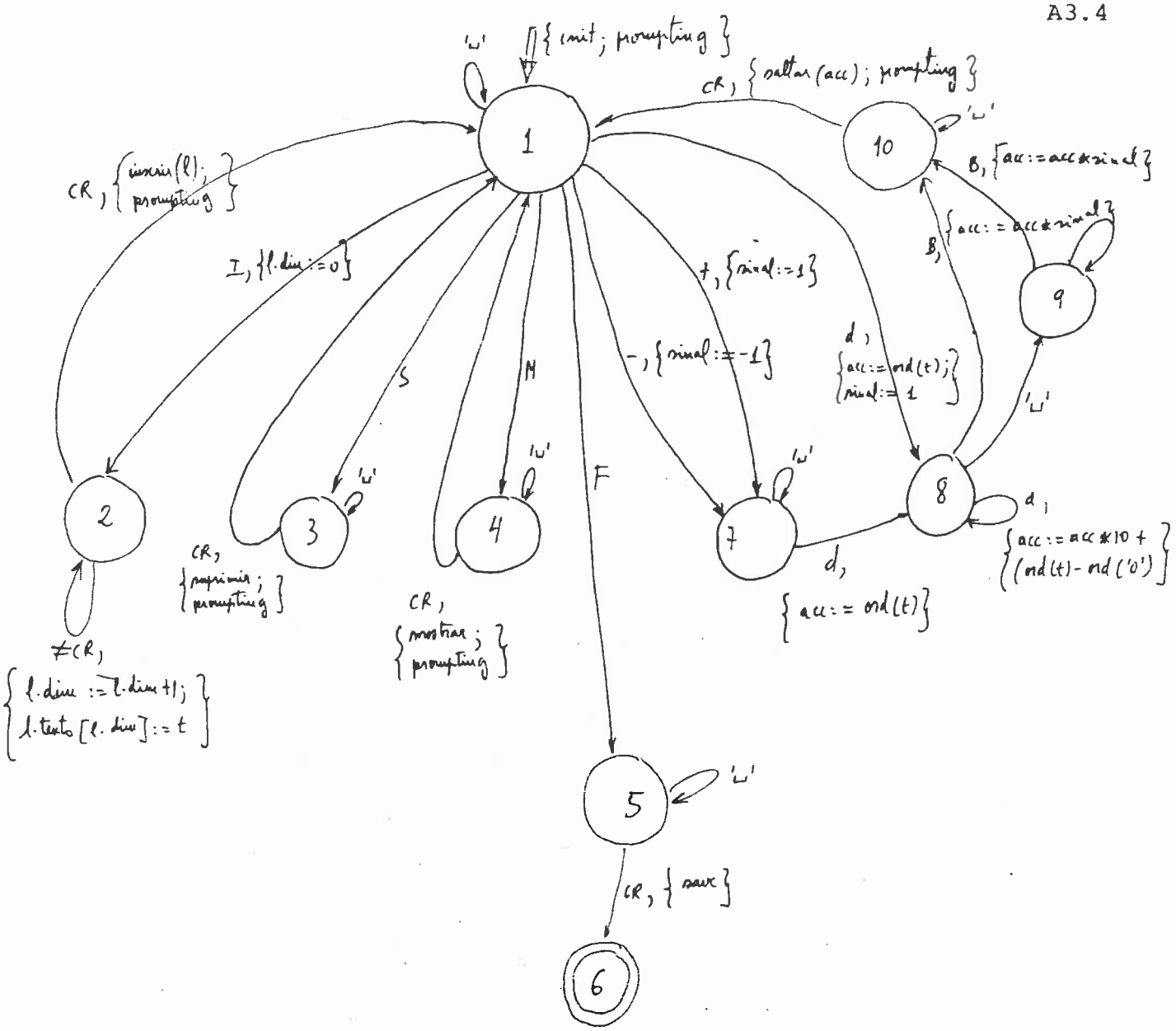
Para explicarmos a semântica e fixar rigorosamente o funcionamento do autómato, tomemos as definições:

```
const CR = chr (13); (*código interno de carriage return*)
conjunto-terminal = char
var t: char; (* caracter corrente lido *)
procedure ler-terminal (var t: char);
(*devolve o caracter lido da consola. Se se atingiu
 o fim da linha devolve t = CR*)
type linha = Record
               Texto: array [1...70] of char;
                 dim: ø..70
             end
var l: linha;
var acc, sinal: integer; (*permitem calcular o inteiro*)
```

Existe também uma estrutura de dados abstracta manipulada pelas operações:

```
procedure init;
    (* lê do input o ficheiro a editar e inicializa
    a estrutura em memória.
    Linha corrente da estrutura toma o valor adequa
    do *);
procedure inserir (l: linha);
    (* insere l antes da linha corrente na estrutu-
    ra em memória.
    A linha corrente não é alterada *);
procedure mostrar;
    (* imprime na consola a linha corrente. Se a es
    trutura está vazia, ou a linha corrente ultrapas
    sa o fim, assinala o facto *);
procedure suprimir;
    (* suprime a linha corrente. A linha corrente
    passa à seguinte. Se não existe assinala esse fa
    cto *);
procedure saltar (n: integer);
    (* modifica a linha corrente em função de n.
    Se ultrapassa o início do ficheiro, ou o seu fim,
    assinala esse facto *);
procedure save;
    (* o conteúdo da estrutura em memória é escrito
    no ficheiro de output *);
```

Tomando em consideração as definições dadas, um primeiro autómato com funções semânticas seria o seguinte:



NOTA: o estado 9 e os lacetes com label '␣' destinam-se a possibilitar que espaços suplementares não desencadeiem imediatamente um erro, dando uma liberdade suplementar ao utilizador, o que quer dizer que a expressão regular que denota a linguagem do editor é ligeiramente diferente da apresentada inicialemente.

Para implementar o automato, por exemplo com case's, tinha-se ainda que definir:

```
type estado = 1 .. 10;
var S: estado;
```

A sua implementação seria trivial.

As acções semânticas repetidas poderiam ser transformadas em procedimentos.

O único aspecto delicado seria o do tratamento dos erros.

Uma forma simples de os tratar seria a introdução de um estado suplementar, 11, que seria o estado erro, assim como um procedimento assinalar-erro (N) que escreveria as mensagens de erro e que era invocado antes da afectação S: = 11;

Assim, em todos os ramos IF ... ELSE IF de cada ramo do case global, haveria sempre um ELSE

```
begin
    assinalar-erro (i);
    S: = 11
end
```

A técnica de tratamento de erros pode ser, por exemplo, desprezar toda a linha corrente até ao CR (caracter de resincronização) e passar ao estado 1.

Um aspecto complementar e interessante de focar é o seguinte:

E se o utilizador dá uma linha para inserir com mais de 70 caracteres ?

Existem 2 formas de resolver o problema:

a) Desprezá-los sem avisar o utilizador passando a acção semantica de memorização no estado 2 à forma:

```
if l. dim <= 69 then
  begin
       l. dim: = l. dim + 1;
       l. texto [ l. dim ] : = t
  end
```

ou então introduzir um estado 2 com a seguinte estrutura:

```
2:
  if t = CR then begin
                      inserir (l);
                      prompting;
                      ler-terminal (t)
                      S: =1
                 end

  else if l. dim <69 then
                 begin
                      l. dim: = l. dim + 1;
                      l. texto [l. dim] : = t;
                      ler-terminal (t);
                      S: = 2
                 end

  else
      begin
           assinalar-erro (...);
           S: = 11
      end;
```

Isto é, aproveitar a sintaxe para tratar os erros semanticos.

ANEXO 4

Problema da árvore de declarações de procedimentos e funções de um programa Pascal

Árvore de declarações de procedimentos e funções de um programa Pascal.

Pretende-se realizar um programa que tem por entrada um programa Pascal sintácticamente correcto e por saída uma árvore das declarações de procedimentos e funções da forma:

id
  id_1
    id_2
    id_3
  id_4
    id_5
      id_6
  id_7
  :

Em que id é o identificador do programa, id_1, id_4 e id_7 identificadores de procedimentos e funções declarados no programa, id_2 e id_3 identificadores de procedimentos ou funções declarados dentro de id_1, id_5 dentro de id_4, id_6 dentro de id_5, etc. (1) .

Para obtermos este programa podemos, numa primeira aproximação, conceber que um determinado módulo transformaria a entrada numa palavra de uma gramática que descreve apenas a informação estrictamente essencial para desenhar aquela árvore. Uma solução, partindo deste pressuposto, é modelizada pelo seguinte módulo:

Módulo árvore-estática define programa

sobre (* id, program, procedure, function, begin, end *)

EQUAÇÕES

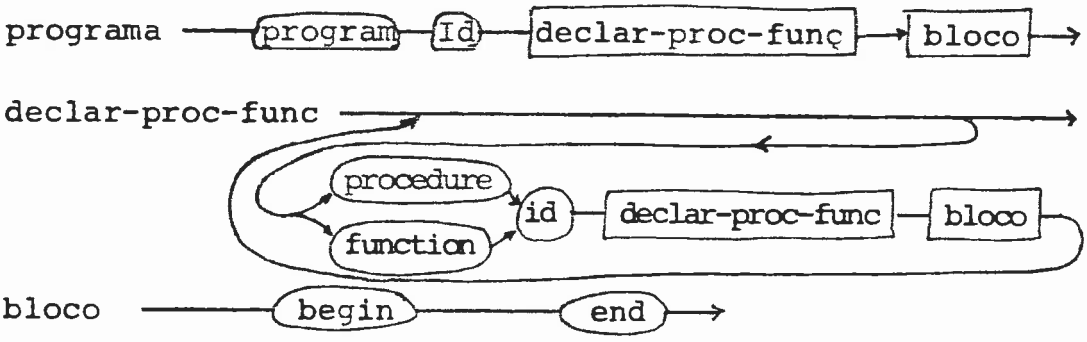
programa = program Id declar-proc-func bloco .

declarac-proc-func =

{(procedure+function) Id declar-proc-func bloco }.

bloco = begin end.

fim árvore-estática.



Que permite verificar imediatamente que se trata de um sistema LL(1).

---

(1) - *Adicionalmente fixa-se que o programa não tem declarações foward, nem parâmetros do tipo procedure ou function numa lista de parâmetros.*

De onde com funções semânticas:

```
Módulo árvore-estática define programa
sobre (* id, program, procedure, function, begin, end
                                                    *)
(* const margem-inicial = 5;
        factor = 10;
   var  pos: integer;
   procedure escrever (palavra: Id);
          begin
               write (' ': pos-1);
               writeln (palavra)
          end;
   procedure incrementar;
          begin
               pos: = pos + factor;
               if pos> 70 then ERRO
          end;
   procedure decrementar;
          begin
               pos: = pos-factor
          end;                              *)
   EQUAÇÕES
   programa = /* pos: = margem-inicial */ program  /* escre
     ver (t); incrementar */ Id declar-proc-func bloco    .
   declarac-proc-func =
         {(procedure + function) /* escrever (Id); incremen
          tar */ Id declarac-proc-func bloco     /* decre-
          mentar */  }.
   bloco = begin end.
   fim árvore estática.
```

Prosseguindo na análise, é necessário conceber um módulo que construa a palavra apresentada à entrada do módulo árvore-estática.

Podemos por exemplo, conceber que esse módulo tem por entrada uma sequência de componentes elementares de um programa Pascal ("tokens") de onde são à partida retirados os comentários e qualquer outro "token" distinto de identificador ou palavra chave . As palavras chave que não são relevantes para o módulo são qualificadas como identificadores. O módulo constrói a palavra que será a entrada do módulo anterior pela função semântica out.

```
Módulo intermediário define programa-int
sobre  (* Id, program, begin, end, procedure, function, case
                                                          *)

(*
 var ficheiro-intermédio: file of  conjunto-terminal - do -
                                   módulo - árvore-estática;
     contador: integer;
 procedure out (t: conjunto-terminal);
         begin
              put (ficheiro-intermédio, t) (*símbolo corrente*)
         end;
                  *)
 EQUAÇÕES

 programa-int = /*reset (ficheiro-intermédio);*/
                program /*out (program);out (t) */Id
                 term1   declarações bloco.
 declarações = { .      /*out (t);*/ (procedure + function)
                 /*out (t);*/ Id  term1  declarações bloco }.
 term1  = {Id}(*diferente de procedure ou function
                                              ou begin*).
 bloco  =  /*out (begin);*/ begin
           /*contador: = 1;
             while contador > 0 do
                begin
                   if (t = begin) or (t = case) then contador
                                              := contador+1
                   else if t = end then contador: =contador-1;
                   ler-terminal (t)
                end;
                out (end);
                        */.
fim intermediário.
```

Repare-se que no exemplo, a semântica é usada para atravessar o bloco. Trata-se de um método espedito de evitar estar a reconhecer toda a estrutura sintáctica das instruções de um bloco, pois para o nosso caso, só nos interessa atingir o end do bloco sem que este seja confundido com o end de um case ou de uma instrução composta.

Finalmente, o último módulo que transformaria o "input" do programa numa palavra sobre o alfabeto do módulo intermediário é um sistema regular ("analizador lexicográfico") que pode ser obtido fácilmente.

Para encontrar uma solução definitiva é necessário estabelecer:

a) Uma forma de acoplamento dos módulos.

b) Em função desta, uma representação para os terminais em jogo.

Para a ligação dos módulos, podemos partir das seguintes ideias:

Sendo o "analisador lexicográfico" um sistema regular, é fácil construir a partir do seu módulo em SINTAX um autómato, e a partir deste um procedimento, que passará a ser o procedimento ler-terminal do módulo intermediário.

Dado que a complexidade do problema é pequena, podem fundir-se os módulos intermediários e árvore-estática num único.

Para tal basta remover de intermediário as acções semânticas de construção do ficheiro intermédio e inserir nele, nas posições correspondentes, as acções semânticas de árvore-estática.

Finalmente o único conjunto terminal em jogo, para além dos caracteres de entrada no analisador lexicográfico, é:

{program, procedure, function, begin, end, case, Id}

o qual se representa bem em:

```
type conjunto-terminal = Record
                           classe: conjunto-classes;
                           valor : packed array [1..max] of char
                           end;
```

onde conjunto-classes =

(TPROG, TPROC, TFUNC, TID, TBEG, TEND, TCASE)

e os testes sobre o terminal corrente passarão a ser feitos sobre t.classe

Para concluir é interessante vereficar como a análise descendente em camadas de gramáticas permite obter as gramáticas mínimas relevantes para o problema, assim como os conjuntos terminais mínimos.