255507-9.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

INSTITUTO DE INFORMÁTICA

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ESPECIFICAÇÃO DE UM SISTEMA DE SUPORTE À IMPLEMENTAÇÃO DE LINGUAGENS

por

CELSO LUIZ LOPES RODRIGUES

Dissertação submetida como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Paulo Alberto de Azeredo
Orientador

Porto Alegre, junho de 1987.

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

CIP - CATALOGAÇÃO NA FONTE

Rodrigués, Celso Luiz Lopes

Especificação de um sistema de suporte à implementação de linguagens / Celso Luiz Lopes Rodrigues. --- Porto Alegre: CPGCC da UFRGS, 1994.

111 p.: il.

Dissertação (mestrado). Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1994.

Orientador: Azeredo, Paulo Alberto de.

Dissertação:Programação:linguagens. Metalinguagens: descrição sintática e semântica. Linguagens de programação: sistemas de suporte à implementação

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Sistema de Biblioteca da UFRGS

N. Registro: 30671

681.32.06(043)

R696E

Autor:

Título: INF

1995/255507-9

Data 1995/06/12

MOD. 2.3.2

AGRADECIMENTOS

Agradeço ao Curso de Pós-Graduação em Ciência da Computação da UFRGS pela oportunidade oferecida, e em especial ao meu orientador, Prof. Paulo Alberto de Azeredo, pela disponibilidade e boa vontade demonstradas.

Agradeço às entidades que me concederam bolsa de estudos ao longo do curso, CAPES e CNPq.

Agradeço também à PUCRS e à Universidade de Caxias do Sul pelas oportunidades de emprego oferecidas quando me faltou a bolsa de estudos, embora estes não sejam fatos ligados intencionalmente, mas que se revelam até hoje experiências valiosas.

Agradeço a todas as pessoas que de alguma forma me ajudaram a concluir este trabalho, seja nas discussões envolvidas, seja no amparo moral e psicológico, e até material, dentre as quais é imprescindível citar as seguintes:

- a Profª Laira Vieira Toscani;
- o colega Tiarajú Asmuz Diverio;
- o colega João Vicente Faria;

SUMÁRIO

LISTA DE SINAIS 6

LISTA DE FIGURAS 7

LISTA DE TABELAS 8

RESUMO 9

ABSTRACT 10

1. INTRODUÇÃO 11

1.1 Motivação 11

1.2 Objetivos do Trabalho 14

1.3 Alguns Sistemas Similares 16

1.3.1 META 16

1.3.2 AED-0 17

1.3.3 GAG 17

1.3.4 PQCC 19

1.3.5 YACC 19

1.3.6 S/SL 21

2. VISÃO GERAL DO SISTEMA SINSEM 23

2.1 Terminologia 23

2.2 Uso do Sistema SINSEM 24

2.3 O Ato de Especificação 27

3. O SISTEMA DE ESPECIFICAÇÃO 29

3.1 O Método Adotado 29

3.2 Estrutura Básica 36
3.2.1 Diagramas de Contorno 36
3.2.2 Sintaxe Abstrata de SINSEM 39
3.3 A Metalinguagem SINSEM 43
3.3.1 Introdução 43
3.3.2 Sintaxe Concreta e Semântica Informal de SINSEM 46
a) Forma geral de uma especificação 47
b) Variáveis de trabalho 49
c) Produções 51
d) Significado de uma produção 55
e) Rotinas Semânticas 61
3.3.3 Semântica Estática em SINSEM 63
3.3.4 Recuperação de Erros Sintáticos 67
3.4 A Construção do Tradutor 69
4. O AMBIENTE DE IMPLEMENTAÇÃO 73
4.1 O Envelope 73
4.2 A Biblioteca Sintático-Semântica 81
4.3 Diagramas deFluxos de Dados deNível Mais Alto 84
5. EXEMPLO DE USO: CSSD 88
6. CONCLUSÕES 106
BIBLIOGRAFIA 108

LISTA DE SINAIS

---> ou <---	atribuição de significado a um nodo de uma árvore sintática
$\overset{n}{\underset{1}{\mathrm{OPER}}}\ i\ \mathrm{OPND\text{-}i}$	O operador OPER é aplicado n vezes (uma para cada operando OPND-i)
$\cup$	união (no conceito da seção 3.2.1)
$\cap$	interseção (no conceito da seção 3.2.1)
$\equiv$	equivalência (no conceito da seção 3.2.1)

LISTA DE FIGURAS

Figura 1.1 Configuração de um verdadeiro sistema de desenvolvimento de linguagens de programação.....15

Figura 2.1 O Sistema SINSEM e o usuário..... 23

Figura 2.2 Uso do Sistema SINSEM..... 25

Figura 3.1 Árvore de uma regra..... 56

Figura 3.2 Árvore de uma regra com significado atribuído à raiz..... 57

Figura 3.3 Significado atribuído a um não-terminal cuja produção correspondente tem um significado atribuído à raiz..... 59

Figura 3.4 Árvore sintático-semântica da produção descrita no texto(regra 20)..... 60

Figura 4.1 Diagrama de sistema do sistema SINSEM.....74

Figura 4.2 DFD's de SINSEM..... 84

LISTA DE TABELAS

Tabela 4.1 Lista de comandos ao envelope do sistema SINSEM 76

Tabela 4.2 Dicionário de dados resumido 87

RESUMO

Neste trabalho é descrita a organização de SINSEM, um sistema de processamento automático de especificações SINtático-SEMânticas de linguagens de programação. É colocada a motivação para um tal sistema, em relação ao uso que se daria a ele. O sistema é situado entre sistemas similares, dos quais alguns são brevemente descritos, incluindo-se exemplos de uso. Procura-se estabelecer conceitos relativos ao trabalho com sistemas de auxílio ao projeto e à implementação de linguagens de programação. É discutida a estrutura lógica do sistema e uma filosofia de utilização, bem como é apresentada a nova metalinguagem proposta para descrição sintática e semântica a qual é um sistema de produções construído a partir da notação BNF, com extensões para se exprimir a semântica de uma linguagem de um modo construtivo, permitindo a especificação de gramáticas livres do contexto sem recursões à esquerda, visando à produção de reconhecedores recursivos descendentes dotados de ações semânticas. São apresentadas a sintaxe concreta e estática da metalinguagem, explicando-se também as funções semânticas (do tipo estático e do tipo concreto) pré-definidas no ambiente de desenvolvimento onde a ferramenta se integra. Tal ambiente pode ser operado por um conjunto de comandos que constituem uma interface (o " envelope "), que inclui também uma biblioteca sintático-semântica (que pode ser atualizada pelo usuário). A interface é descrita por meio de DFD's e de um dicionário de dados. Ao final é apresentado um exemplo, parcialmente comentado (capítulo 5), de definição completa de uma linguagem de programação (CSSD), usando-se a metalinguagem proposta e os itens pré-definidos da biblioteca sintático-semântica.

PALAVRAS-CHAVE: programação: linguagens, metalinguagens, descrição sintática e semântica, linguagens de programação: sistemas de suporte à implementação.

TITLE: "SPECIFICATION OF A SUPPORT SYSTEM TO LANGUAGE IMPLEMENTATION".

ABSTRACT

In this work it is specified the constitution of SINSEM, an automatic processing system for SYNtactic and SEMantic specifications of programming languages. It is discussed motivation for such a system, according to uses it could have. The system is situated among similar systems, and some of these are briefly described, including examples. It is attempted to establish concepts about working with ,design and implementation aiding systems for programming languages. It is discussed the logical structure of the system and a philosophy of use, as well is presented a new metalanguage proposal for syntactic and semantic descriptions that is a production system constructed from BNF notation, with extensions to give language semantics in a constructive way, allowing specification of free context grammars with no left recursions, aiming the generation of preliminary versions of recursive descent parsers with semantic actions. The abstract and concrete syntax of the metalanguage are presented, and are also explained the semantic functions (of static and concrete type) predefined with and within the development environment where this frame is inserted. That environment can be operated by a set of commands that constitutes an interface (the " envelope") wich includes too a syntactic-semantic library (this can be made up to date by the user own). The interface is described by means of a set of DFD's and a data dictionary. At the end, it is presented an example, partially commented (chapter 5), of a complete definition of a programming language (CSSD), using the proposed metalanguage and the predefined items of the syntactic-semantic library.

KEYWORDS: programming: languages, metalanguages, syntactic and semantic description, programming languages: implementation support systems.

1 INTRODUÇÃO

1.1 Motivação

Em um meio computacional, tem papel destacado a função de comunicação entre os diversos sistemas e o usuário. O grau de complexidade e eficiência dessa comunicação varia conforme uma série de fatores, entre eles o da natureza das linguagens de programação disponíveis, o que motiva continuamente o projeto de novas linguagens, adaptadas a problemas e ambientes específicos. Os ambientes vão desde os mais técnicos, onde os usuários serão especialistas em computação, até os mais leigos, onde os usuários desejam apenas valer-se de uma forma a mais imediata possível do computador, mas não podem prescindir de algum grau de programação. Este segundo caso ocorre, por exemplo, quando, pela especificidade do tema, um programador profissional encontra dificuldades para programar as soluções desejadas ou, noutro exemplo, quando o usuário enfrenta uma variedade de itens de problema, tornando-se necessário prover a ele uma série mais ou menos grande de rotinas, uma para cada item.

Um conjunto de rotinas dedicadas a uma área de problemas vem a constituir o que geralmente se denomina "pacote". Para seu uso é desenvolvida uma linguagem para especificação de problemas e solicitação de soluções. Algumas dessas linguagens são simples, e suas sentenças lembram um mero preenchimento de formulário burocrático, onde são fornecidos dados e se requer providências. Exemplo disso é a linguagem de controle do pacote estatístico SPSS. Outras propostas evoluem mais, como por exemplo as linguagens dos pacotes SAS (estatística) e CSSP (simulação de processos). Estas incluem tipos de dados simples e algumas estruturas de controle.

Já uma linguagem desenvolvida para atender à especificidade de uma área, incorporando suas abstrações típicas, especialmente os tipos de dados, suas estruturas e operações, resulta no que se denomina "linguagem orientada para o problema". Exemplos destas são LORANE (analise estrutural) e GAMBIT (desenvolvimento de jogos eletrônicos).

Um enfoque adicional quanto à especialização de linguagens de programação é dado pela sua adequação a um determinado ambiente de trabalho, quando então é dita "orientada para um ambiente". Exemplos se encontram em experiências na área de programação de sistemas interativos, sistemas em tempo real, de automação de escritórios, de programação de máquinas de arquitetura não-von Neumann, etc.

Como já foi dito, um especialista em computação pode não ser conhecedor profundo de um assunto no âmbito do qual lhe for solicitado o projeto de um sistema de programas. Se a solicitação desse usuário for bem específica, ocasional, e o sistema resultante for considerado "grande" e/ou tenha prevista uma vida útil pelo menos razoável, se não longa, então será viável (economicamente) a dedicação do especialista na sua consecução, isoladamente ou mesmo em equipe, investindo daí no aprendizado em questão.

Por outro lado, se as solicitações do usuário tendem a ser freqüentes e de pequenos programas, que via de regra são executados apenas uma vez e que, ademais, diferem apenas em pequenos detalhes entre si, logo se optará pelo estabelecimento de um conjunto de rotinas parametrizadas que, mais um passo, podem tornar-se um pacote. Este passo pode ser dado com a implementação de uma linguagem de controle.

Com o avanço das técnicas e metodologias da área o pacote tende à obsolescência. Assim é que um estágio seguinte se alcançará ao se projetar uma linguagem orientada para o problema dessa área, o que será uma ferramenta mais poderosa e permanente. Note-se que essa evolução não é determinante nem necessária, quer dizer, pode-se evidentemente partir desde o início em busca de uma linguagem de programação e não de um pacote para uma certa área.

Finalmente, mesmo em se tratando de linguagens de programação ditas "de propósitos gerais", está-se permanentemente pesquisando novas formulações para elas, onde se experimentam propostas relativas aos tipos de dados suportados, às estruturas de controle fornecidas, aos regimes de execução (seqüencial, concorrente, paralelo, distribuído), ao potencial expressivo de uma ou outra sintaxe em particular, às facilidades para suportar uma ou outra filosofia de construção de sistemas de programação, ao modo de especificação dos programas (imperativo, funcional, lógico-formal), etc.

Enfim, todas essas situações distintas de uso de linguagens de programação justificam a pesquisa de novas linguagens e a nova torre de Babel, como sugere a conhecida capa da revista "Computer Languages", cresce muito mais rapidamente e crescerá por muito mais tempo que a antiga, talvez para sempre.

E' num tal contexto que ganha utilidade um sistema auxiliar ao desenvolvimento de linguagens de programação, o qual se propõe a prover meios, ao projetista, de obter protótipos operacionais de sua linguagem, de

modo a poder avaliá-la em vários aspectos, precedendo uma implementação possivelmente mais eficiente.

1.2 Objetivos do Trabalho

Inicialmente, o objetivo era implementar um "sistema de auxílio ao desenvolvimento de linguagens de programação" (fig.1). Entretanto, um tal sistema deveria prover ferramentas que incrementassem a facilidade da tarefa de desenvolvimento de uma linguagem a partir já da sua concepção, desde a identificação das estruturas que a linguagem deveria conter até a melhor sintaxe para representá-las, produzindo-se daí uma especificação formal. Esta especificação serviria como base para a produção automática de um tradutor, através de um gerador de tradutores. Esta segunda fase (implementação) tem sido estudada em grande extensão, existindo também muitos sistemas de auxílio à implementação (SAI's). A fase de concepção conta com poucos estudos, dirigidos para casos particulares (ADA, por exemplo /ICH79/), sendo pequeno também o número de trabalhos sobre os princípios gerais para o projeto de linguagens de programação (por exemplo, /WIL77/, /MEE81/, /GHE85/) e aparentemente inexistem propostas acabadas para sistemas de auxílio ao projeto de linguagens de programação (SAP's).

Conseqüentemente, a produção de um sistema de auxílio ao desenvolvimento de linguagens (SAD) exigiria uma extenso embasamento e só poderia ser encaminhado por uma subdivisão do trabalho em vários projetos.

Submetemo-nos, então, a algumas restrições. A primeira delas reduz o sistema a ser produzido a um SAI, capaz de processar uma

especificação adrede derivada de uma atividade manual de projeto. Este sistema inclui:

a) um gerador de tradutores (de fato o núcleo do sistema);

b) uma interface de comunicação e gerenciamento do gerador.

Na fase de desenvolvimento em que este trabalho se encontra, sua parte mais significativa é o estabelecimento de um formalismo para a especificação da linguagem a ser implementada. Esta é a metalinguagem associada ao gerador.

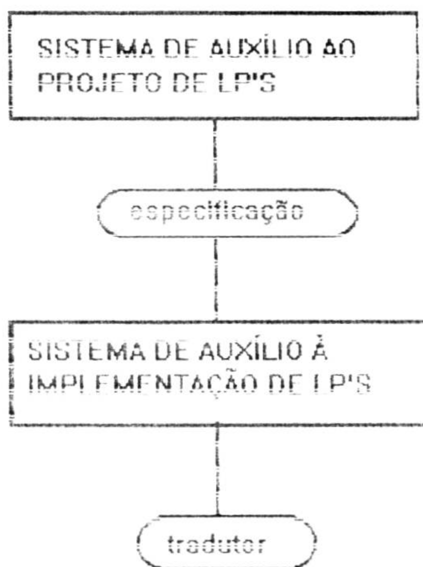


Figura 1.1 Configuração de um verdadeiro sistema de auxílio ao desenvolvimento de linguagens de programação.

Assim é que este documento relata o trabalho desenvolvido. No capítulo 2 esboça-se a estrutura geral do sistema proposto. O capítulo 3 descreve o sistema de especificação adotado, ou seja, a metalinguagem acima citada, com seus mecanismos. Ainda no capítulo 3 é abordada

uma proposta para o processo de transformação da especificação em um tradutor correspondente. No capítulo 4, é dada uma descrição do que é o "ambiente de implementação", formado pela interface de comunicação (o "envelope") e alguns outros dispositivos. O capítulo 5 traz exemplos de uso da metalinguagem proposta. As conclusões estão no capítulo 6.

1.3 Alguns Sistemas Similares

1.3.1 META

Dentre os sistemas citados por /FEL68/, um dos mais antigos é META, que já aceitava a descrição sintática numa forma próxima à BNF, mas exigia que as rotinas mais elementares ao nível léxico fossem fornecidas (como para processar identificadores), da mesma forma que as semânticas. Um exemplo de produção na sua linguagem seria: /FEL68/

BPRIMARY = .ID .OUT('LD' *) ! '(' UNION ')'

que especifica que um item primário booleano (BPRIMARY) seria reconhecido se fosse reconhecido um identificador (ID) ou uma construção UNION entre parênteses (UNION é outra regra da gramática). O ponto identifica (como em .OUT) as rotinas fornecidas a posteriori pelo usuário. O '*' substitui uma referência à última construção reconhecida (no caso, ID). A rotina OUT do exemplo dá saída no código objeto 'LD <id>'. Este sistema foi desenvolvido por Schorre, em 1964. A versão META II já dava ao usuário algumas rotinas pré-definidas, como ID, LABEL e EMPTY. Os reconhecedores produzidos são do tipo recursivo descendente e não incluem recuperação de erros.

1.3.2 AED-0

Esta é outra proposta (Ross, 1966) também citada em /FEL66/. Está baseada na extensibilidade de linguagens de programação, via macroinstruções em alto nível. Um exemplo é o seguinte:

```
DEFINE MACRO LOOP(P1, P2) TOBE
FOR P1 := 1 STEP 1 UNTIL P2 DO    ENDMACRO
```

O comando especificado teria um uso como em

```
LOOP( I, N )  A[ I ] := 0
```

A linguagem-base de AED-0 é ALGOL e com ele se pode produzir macrolinguagens da "família" AED-n.

1.3.3 GAG

Os exemplos a seguir são mais modernos. GAG (acrônimo para 'Generator Based on Atribute Grammars') /KAS82/, usa como metalinguagem a gramática de atributos, expressa numa linguagem de sintaxe especial, aplicativa (ALADIN), em que se pode declarar tipos (contando com inteiros, listas, terminais, não-terminais e atributos herdados e sintetizados), permitindo-se a especificação de funções semânticas e mensagens de erro. A sintaxe da linguagem sob definição é dada em BNF estendida com alguns operadores como [e] (para opcionais). A semântica é

dada pelas funções, que podem ser especificadas com atribuições e expressões condicionais estruturadas com "if-then-else" e "case". Uma série de funções pré-definidas é oferecida para se trabalhar com listas, além de duas especiais para se produzir código objeto (uma para inteiros, outra para caracteres). Os não-terminais são declarados conjuntamente com seus atributos, os quais devem ser ativados conforme participem ou não de uma certa produção ao ser referido o não-terminal.

Da referência de que dispomos não foi possível extrair um exemplo curto e compreensível, dada a multiplicidade de estruturas da linguagem. O que apresentamos aqui vem da especificação da linguagem Pascal, a qual ocupa cerca de 70 páginas:

```
rule r123:
   statement ::= var-denot ':=' expr
STATIC
statement.at-labels := tp-labels();
CONDITION
   f-assignment-compatible(var-denot.at-type,
                  f-reduce(expr.at-type))
  MESSAGE "INCOMPATIBLE TYPES IN ASSIGNMENT"
END;
```

Em linhas gerais, a regra r123 define o comando de atribuição. A lista vazia (de elementos do tipo tp-labels) é atribuída ao atributo at-labels do não-terminal statement. Uma condição é verificada com a função

f-assignment-compatible, de resultado booleano. Seus argumentos são o atributo at-type de var-denot e o resultado da aplicação da função f-reduce ao atributo at-type de expr. E' estabelecida uma mensagem para o caso da condição não ser satisfeita. A referência não elucida o tipo de reconhecedor produzido.

1.3.4 PQCC

PQCC (acrônimo para 'Production Quality Compiler Compiler')/WUL80/ é um gerador que produz compiladores fortemente direcionados para a otimização de código ("expert generators"), para o que aplica técnicas da inteligência artificial (análise de meios e fins). A metalinguagem usada é calcada na correção de programas, baseando-se nos axiomas de Hoare, o que a faz obscura e difícil. Permite descrever, no entanto, as estruturas da linguagem fonte e as instruções da linguagem objeto, de modo a se obter conjuntamente um interpretador para esta última, o que torna este sistema singular entre outros.

1.3.5 YACC

YACC (acrônimo para 'Yet Another Compiler Compiler') /JOH78/é um gerador de reconhecedores sintáticos dotados de ações semânticas (rotinas em liguagem C), ascendente, com um símbolo de "look-ahead". Trabalha em conjunto com um gerador de analisadores léxicos LEX. Sua linguagem de descrição é uma forma de BNF. Provê facilidades para recuperação de erros e especificação da precedência dos operadores.

O exemplo a seguir é adaptado de /JOH78/. Supõe-se que a rotina datew(a. b. c) tome mês. dia e ano. nesta ordem. e produza o dia da semana em que tal dia cai. A data pode ser entrada como July. 4. 1776 ou 7/4/1776 ou 4 July 1776.

```
%token DIGIT MONTH
%%
input: /* arquivo vazio é legal */
  | input date '\n'
  | input error '\n' { /* ignore linha caso erro */ }
date: MONTH day ',' year
    { datew( $1. $2. $4 ): }
  | day MONTH year
    { datew( $2. $1. $3 ): }
  | number '/' number '/' number
    { datew( $1. $3. $5 ): }
day: number
year: number
number: DIGIT
  |  number DIGIT
    { $$ = 10 * $1 + $2: }
```

No trecho acima. primeiramente são declarados os identificadores dos "tokens" que se espera venham codificados pelo analisador léxico (gerado por LEX)(no caso. 0 a 9 para DIGIT e 1 a 12 para MONTH). Em cada sintaxe válida para date. o efeito é chamar a rotina

datew, que calculará o dia da semana. A regra para input significa o acesso ao arquivo fonte e se for obtida uma data com formato inválido, este é atribuído a error. Na regra number, o efeito semântico é calcular o valor decimal do número.

Uma curiosidade acerca de YACC é que seu acrônimo, em alguns círculos, tornou-se um substantivo comum, significando qualquer gerador de compiladores.

1.3.6 S/SL

S/SL (Syntax/Semantic Language) é uma experiência um pouco diferente /HOL82/. Trata-se de uma linguagem orientada para escrever compiladores. Ela não tem tipos. Suas estruturas de controle são seqüência, repetição e seleção. Conta com comandos para ler, comparar e dar saída a "tokens", assinalar erros, invocar sub-rotinas (chamadas "regras") em S/SL mesmo ou numa linguagem hospedeira, como Pascal.

O exemplo a seguir vem de /HOL82/:

```
Operators:
{ [ | add: @CheckInteger @CheckInteger TypePush( int)
  | intersect: @CheckBoolean @CheckBoolean
          TypePush( bool )
  | equal: @CheckEquality TypePush( bool )
  | * : > ] }
```

Uma invocação como @CheckBoolean significa uma outra rotina em S/SL (como Operators). TypePush é uma rotina em Pascal. A construção { } equivale a uma repetição infinita, que pode ser interrompida pela operação '>' ("exit"). O asterisco ('*') vale por qualquer outra opção (é o "otherwise") da construção [| ... | ... ...] que é uma seleção. Suas opções, como no exemplo, testam o último "token" acessado. A cada repetição do laço { } um "token" é buscado da fonte, usando-se para tal uma tabela que terá sido fornecida por uma declaração como

input: add '+' equal '=' assign ':=' ... etc ...

A tabela de saída também é fornecida:

output: int add jmp ... etc ...

Alguns "tokens" são pré-definidos, como integer, empty, etc.

Programas em S/SL são "compilados" em um conjunto de tabelas que são reprocessadas a cada execução por um programa chamado "table walker", que seria o seu interpretador, não havendo assim, propriamente, a geração de um programa compilador. Segundo os autores, S/SL pode simular uma máquina de Turing.

2 VISÃO GERAL DO SISTEMA SINSEM

Conforme já mencionou-se na Introdução (item 1.2), SINSEM (SINtaxe e SEMântica) é um sistema de auxílio à implementação de linguagens de programação. Ele consta basicamente de um processador de símbolos dirigido por sintaxe, o qual é "envolvido" por um gerenciador que se encarrega de prover a "interface amigável" com o usuário. Este gerenciador, doravante, será designado como o "envelope". Uma representação pictórica disso é dada na figura 2.1, e será mais detalhada na seção 4.1.

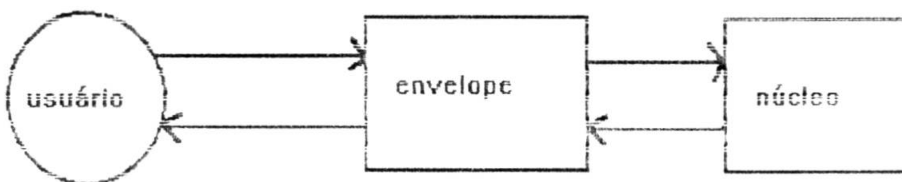


Figura 2.1 O sistema SINSEM e o usuário

2.1 Terminologia

Na apresentação a seguir, há referências de forma especial a uma ou outra linguagem de modo a caracterizá-la quanto à função que ela desempenha num dado sistema processador de linguagens. Linguagem de uso é aquela linguagem que se quer ver implementada e que vai ser então alvo de uma especificação. Linguagem de especificação ou metalinguagem é a linguagem usada para descrever, definir, ou, enfim, especificar a linguagem de uso, de qualquer ponto de vista (sintático, semântico ou mesmo outros). Linguagem de trabalho é a linguagem em que se pretende implementar o tradutor para a linguagem de uso. Linguagem objeto é aquela usada para exprimir o resultado da tradução de uma linguagem de uso.

De modo análogo, especificações são escritas em metalinguagem, programas-fontes são escritos em linguagem de uso e programas-objetos vão estar em linguagem objeto. Normalmente se usará identificar completamente o caso, mas, em geral, programa significará programa-fonte.

Quando se falar em tradutor poder-se-á estar referindo a tradutor propriamente dito ou a interpretador /LEC82/ . No primeiro caso estão programas que tomam programas escritos em uma linguagem e o traduzem em outra. No segundo, estão programas que executam as ações especificadas nos programas-fontes à medida que os vão analisando sintáticamente. Entre os tradutores, pode-se distinguir os montadores, que traduzem de linguagem de baixo nível ("assembly languages") para linguagem de máquina; os compiladores, que traduzem de linguagem de alto nível para linguagem de baixo nível (ou mesmo, mais raramente, para linguagem de máquina), e os pré-processadores, que traduzem de uma linguagem de alto nível para outra (quase sempre aquela é uma extensão desta).

2.2 O Uso do Sistema SINSEM

O usuário do sistema SINSEM será um interessado em implementar uma nova linguagem de programação, para o que fornecerá ao sistema uma especificação da mesma, escrita na metalinguagem SINSEM. O próprio ato de escrever a especificação (uma vez convenientemente projetada a linguagem) é auxiliado pelo sistema: a especificação é composta durante o que se denomina uma "sessão de definição". Para tanto, o usuário

conta com o auxílio do envelope, que o guiará nessa atividade de escrever a especificação.

O sistema recebe, então, a especificação de uma linguagem de uso e produz um tradutor para ela, que será um montador, compilador ou interpretador, conforme seja a especificação.

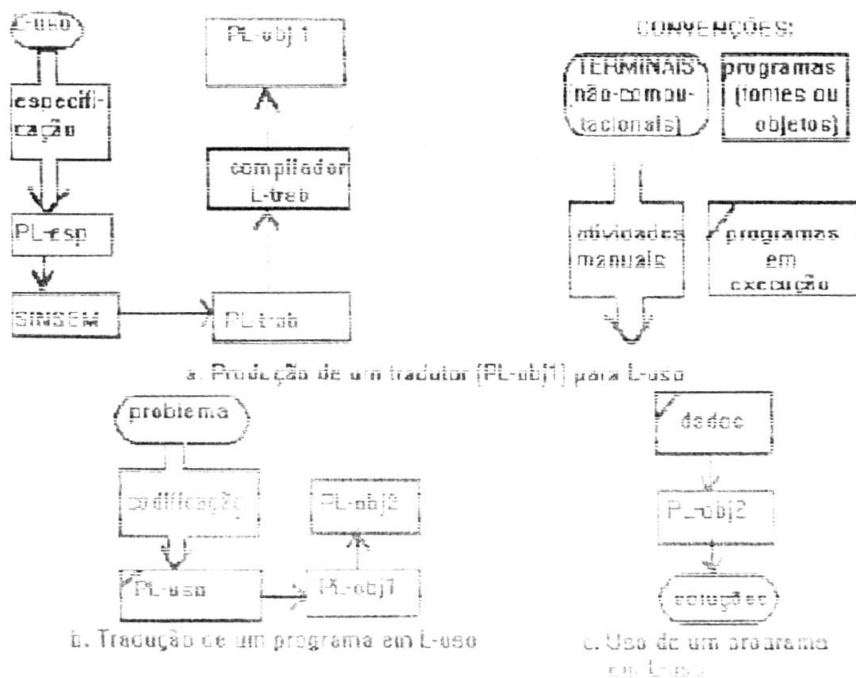


Fig. 2.2 Uso do Sistema SINSEM

Um esquema desse processo pode ser visto na figura 2.2. A linguagem de uso L-uso, depois de ser projetada, passa a ser representada por uma especificação, PL-esp, que pode ser entendida como um "programa" para o sistema SINSEM, conduzindo-o a produzir um tradutor, PL-trab, expresso em um linguagem de trabalho (por exemplo, Pascal). PL-trab, submetido como programa-fonte a um compilador para essa linguagem de

trabalho. gera um programa-objeto PL-obj1. que é uma versão executável do tradutor PL-trab.

Os usuários de L-uso vão codificar programas como PL-uso. que serão programas-fonte para o tradutor PL-obj1. Cada programa PL-uso originará programas executáveis PL-obj2 (resultantes de uma compilação/montagem) ou poderão ser diretamente "executados". isto é. interpretados. se o tradutor for do tipo interpretador (não é o caso representado na figura. mas é facilmente perceptível isto).

Os usuários de L-uso vão codificar programas como PL-uso. que serão programas-fonte para o tradutor PL-obj1. Cada programa PL-uso originará programas executáveis PL-obj2 (resultantes de uma compilação/montagem) ou poderão ser diretamente "executados". isto é. interpretados. se o tradutor for do tipo interpretador (não é o caso representado na figura. mas é facilmente perceptível isto).

A diferenciação entre PL-obj1 e PL-obj2 significa não só o fato de serem programas diferentes (o primeiro é o tradutor de L-uso e o segundo é um programa-objeto resultante de uma tradução de L-uso) mas também de poderem estar em linguagens objetos diferentes. ou seja. poderem ser executados em máquinas diferentes.

2.3 O Ato de Especificação

Sob este título, pretende-se propor uma postura do usuário frente ao sistema SINSEM em função da sua linguagem de uso com vistas a obter um tradutor.

Apesar de a especificação se fazer em uma linguagem de alto nível, exige-se do especificador uma cultura mínima em termos de implementação de linguagens de programação. Ele deve conhecer:

a) o método de análise sintática recursivo descendente;

b) as expressões na sua linguagem objeto correspondentes às estruturas que ele pretende componham sua linguagem de uso (os assim ditos "esqueletos de código");

c) as estruturas de dados típicas de um tradutor, como tabelas de símbolos, por exemplo, e sua manipulação (tais estruturas aparecem, nas especificações para o sistema SINSEM, implementadas na forma de pilhas (na verdade, pilhas-vetores, já que admitem acesso, mediante algumas operações pré-definidas, a seus elementos intermediários; veja a seção 3.3.3) que é o tipo de dado agregado fornecido pelo sistema).

Para o especificador, o programa-fonte a traduzir deve ser visto como um "string" de terminais que corresponde à fronteira da árvore sintática. Tal árvore é construída de cima para baixo, a partir da raiz, de uma forma virtual, através das aplicações das regras sintáticas responsáveis pela descrição dos diversos "substrings" que compõem o programa.

O programa-objeto deve ser visto como um outro "string", o qual vai sendo construído progressivamente, à medida que vão se verificando corretas as hipóteses adotadas pela análise sintática. As hipóteses são as regras sintáticas. A construção desse "string-objeto" é providenciada pelas rotinas semânticas encarregadas da geração de código, sempre que uma regra é testada com sucesso e a ela tenha sido atribuído algum significado no contexto onde tenha sido referida. Este ponto de vista deve ficar mais claro após o capítulo 3.

3 O SISTEMA DE ESPECIFICAÇÃO

Neste capítulo será descrita a linguagem de especificação do sistema SINSEM. Para caracterizá-la melhor, esta apresentação é antecedida de uma discussão que procura dar notícia da sua inspiração e da sua localização entre outras metalinguagens.

3.1 O Método Adotado

O sistema SINSEM se propõe uma ferramenta de uso fácil, intuitivo, ao menos para os usuários que contem com uma cultura mínima na área de estudo e desenvolvimento (compreendendo a implementação) de linguagens de programação. Como base do processo de especificação, a metalinguagem deveria apresentar tais qualidades, e foi pensando nisso que se optou não por criar um formalismo inteiramente novo, mas sim por estender algum já existente e bem conhecido, opção esta já adotada no projeto de outros sistemas similares /FEL68/, /KAS82/. O formalismo escolhido, a Forma Normal de Backus (BNF), é o mais conhecido e empregado, no seu formato original ou no de uma grande família de variações /WIL82/, para especificar-se a sintaxe de linguagens de programação.

Quanto à semática, as diversas abordagens do problema da definição formal se refletem nas metalinguagens propostas. Duas tendências globais se verificam, com objetivos distintos, conforme /HOA74/. Uma, a das definições <u>implícitas</u>, se embasa na formalização de propriedades dos programas, através de regras dadas na forma de axiomas lógicos. Esta

classe visa principalmente a comunicação com o usuário. A outra, a das definições construtivas, propõe definições que são mapeamentos entre estruturas da linguagem em definição e as estruturas de algum outro "corpus" semântico considerado "fundamental", ou, pelo menos, independente da linguagem em definição. As definições construtivas se destinam mais aos implementadores.

A definição construtiva pode ser compilativa, quando o mapeamento se dá sobre alguma notação formal, como lambdacalculus, por exemplo, ou interpretativa, se o mapeamento é sobre um conjunto de procedimentos que modelam as ações especificadas pelos programas a serem escritos na linguagem em definição (linguagem de uso). Exemplo de definição implícita é a semântica denotacional /TEN76/, enquanto um exemplo de linguagem formal para definições construtivas é VDL (interpretativa) /WEG72/.

A metalinguagem aqui apresentada segue a filosofia construtiva interpretativa e visa prover meios de descrever a sintaxe e a semântica de linguagens de programação, sendo passível de procesamento automático, de modo a se poder produzir a partir de uma definição de linguagem um tradutor para a mesma (compilador,interpretador ou montador). Ainda para situá-la melhor, pode-se observar que, como a definição da semântica se faz através de alguns mecanismos de extensão a um formalismo meramente sintático (BNF), o SAI correspondente qualifica-se como um processador de símbolos dirigido por sintaxe, em oposição a outros tipos de sistemas, como os assim chamados compiladores de compiladores, os quais se caracterizam por receber a definição da sintaxe em separado da definição semântica, com formalismos próprios para cada fase /FEL68/.

Prosseguindo com a caracterização preliminar do sistema de especificação, há que se notar a ocorrência de dois tipos de semântica. Um, aquele dito _semântica dinâmica_, que se relaciona com o comportamento de u'a máquina correspondente a uma dada sentença de um programa. O outro, a _semântica estática_, se relaciona com a observância de regras de interrelações entre si de trechos diferentes do programa. Tais regras também são chamadas regras sensíveis ao contexto. Exemplos de regras da semântica estática são os seguintes:

1. os argumentos na invocação de um sub-programa devem concordar em número, tipo e ordem com os parâmetros da correspondente definição;

2. uma variável só pode ser usada se tiver sido previamente declarada, no bloco em que aparece ou em algum que o contenha;

3. numa referência a um "array", seus subscritos devem ser tantos quantas forem as dimensões especificadas na sua declaração /WIL78/.

Observe-se que tais itens não podem ser deduzidos de uma especificação sintática comum (BNF) e tampouco fazem parte do _significado_ propriamente de uma sentença _válida_ conforme a especificação sintática:

uma sentença pode ser válida sintaticamente e violar regras da semântica estática. O significado é dado pela semântica dinâmica.

Em geral, a semântica estática é estabelecida informalmente, como acima. Isto traz problemas ao implementador, que segue a descrição sintática ao programar o tradutor e, em meio à análise sintática, precisa atender às regras da semântica estática sem dispor de um guia formal.

A metalinguagem do sistema SINSEM provê, através de um mecanismo comum a ambos os casos, meios de especificar a semântica estática e a semântica dinâmica.

Em relação à semântica ainda se pode classificar as descrições em duas classes /WIL81/ : aquelas dos formalismos que usam não-terminais generalizados e aquela dos formalismos que apenas associam significados a não-terminais ou a produções. No primeiro caso estão os seguintes exemplos: sistemas canônicos de Ledgard e Donovam /DON72/ e as gramáticas de dois níveis de van Wijngaarden /MAR76/; no segundo caso, os exemplos, gramáticas de atributos de Knuth /MAR76/ e BNF's estendidas na forma de esquemas de tradução dirigidos por sintaxe (SDTS: syntax directed translation scheme /AHO72/), como a proposta de /WIL80/. E' neste segundo grupo que se situa a metalinguagem aqui apresentada. Conforme /WIL81/, estes formalismos são os que mais se adaptam a uma especificação com dificuldades da semântica dinâmica e estática, a par de guardar a necessária clareza de uma especificação sintática, dirigindo-se ambas para a documentação e para a implementação da linguagem de uso sob definição.

A seguir são dados exemplos de uso de alguns dos formalismos citados acima.

a) SD - Semântica Denotacional

Um comando de desvio incondicional com rótulo invariável (o exemplo é de /TEN76/):

$$\mathcal{C}[\![\text{go to } I]\!]\sigma\theta\rho \quad = \quad (\rho\ [\![I]\!]\ \mid C\ )\sigma$$

isto significa que o programa a seguir ao "go to" é $\theta\sigma$ e ρ é o trecho que o antecede. A continuação θ é ignorada e o controle é transferido para o trecho atribuindo-se ao contador de programa C (ou "memória corrente", como é chamado em SD) o valor do rótulo I. Observe-se a quantidade de símbolos especiais e diferentes alfabetos necessários para expressar diferentes tipos de operadores. E' verdade que com isso a SD consegue ser muito sucinta, mas certamente não se presta para o processamento automático.

b) VDL - Vienna Definition Language

Um comando de atribuição é dado em VDL através das seguintes regras (o exemplo é adaptado de /MAR76/).

PARTE SINTÁTICA

```
[ sin-1 ] is-c-asgt-stm = ( <s1: is-c-id>,
                            <s2: is-':='>,
                            <s3: is-c-exp> )
```

isto significa que um comando de atribuição (asgt-stm) tem 3 componentes, s1, descrito por is-c-id; s2, a constante sintática ':=' ; s3, descrito por is-c-exp.

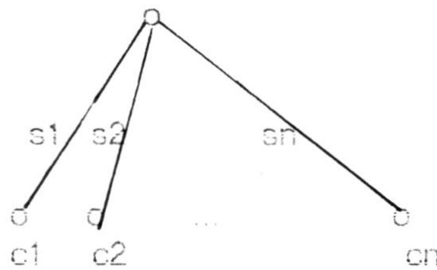
Continuando a definição. o identificador à esquerda de ':='. por exemplo. é dado por

[sin-2] is-c-id = (<s1: is-c-letter>. ...)

As reticências significam a repetição um número indefinido de vezes do item anterior. Os itens s-i são chamados "seletores". e as fórmulas "is-" funcionam como predicados que são verdadeiros se o objeto ao qual se aplicam tiverem todos os "componentes" dos quais os s-i são prefixos. Assim. um objeto sintático é visto como um registro

(<s1: c1>. <s2: c2>. ... <sn: cn>)

ou como uma árvore:



O componente c-i de um objeto t pode ser extraído mediante a fórmula funcional s-i(t).

PARTE SEMÂNTICA

[sem-1] interpret-assignment(t) =

assign(s1(t). value):

value: eval-exp(s3(t))

[where is-c-id(s1(t)) and

is-c-exp(s3(t))]

Isto significa que a execução interpretativa de um comando de atribuição (no caso, representado pelo parâmetro t) equivale a atribuir value ao componente s1 de t (o receptor), sendo que value é dado pela aplicação de eval-exp ao componente s3 de t (o emissor).

[sem-2] assign(target, value) =

μ (store(ξ):<target: value>)

A operação μ (A: <s1: B>) resulta no objeto A com seu componente de seletor s1 substituído por B. Do exemplo acima inferimos então que existe um objeto ξ (obs: é o estado da máquina de execução da linguagem sob definição; deve ser definido com um predicado adequado), o qual tem um componente store, o qual tem um componente target. Este último recebe value quando da interpretação de um comando de atribuição.

Como se vê, VDL se aproxima bastante de um linguagem de programação, mas com uma notação não muito clara, apesar de permitir composições bem complexas.

O motivo por que se adota aqui a proposta construtiva interpretativa é dar simplicidade à metalinguagem, ao mesmo tempo sem exigir grande esforço de aprendizado do usuário, baseando-a em um formalismo popular como é BNF, estendendo-o com poucas construções, deixando a parte de especificação semântica relacionada com uma linguagem de programação (Pascal). Além disso, o formalismo torna-se mais fácil de ser processado automaticamente.

3.2 Estrutura Básica

Nesta seção abordar-se-á a organização mais geral de uma especificação de uma linguagem em SINSEM. Em outras palavras, o que se estará considerando será sua sintaxe abstrata, isto é, os componentes de cada estrutura da especificação, sem estabelecer a forma como os mesmos seriam representados numa implementação, o que constituiria sua sintaxe concreta /WEG72/.

3.2.1 Diagramas de Contorno

O recurso que será utilizado para esta apresentação será o diagrama de contorno, um meio gráfico usado para melhorar a visualização de um conjunto de módulos aninhados hierarquicamente. De um modo mais formal, a mesma descrição poderia ser feita via uma linguagem como VDL, por exemplo, em que se veria esta organização representada por uma árvore. O diagrama de contorno foi escolhido por ser de mais fácil apreensão, além de propiciar uma representação analógica: pode ser aplicado diretamente sobre o texto de uma especificação.

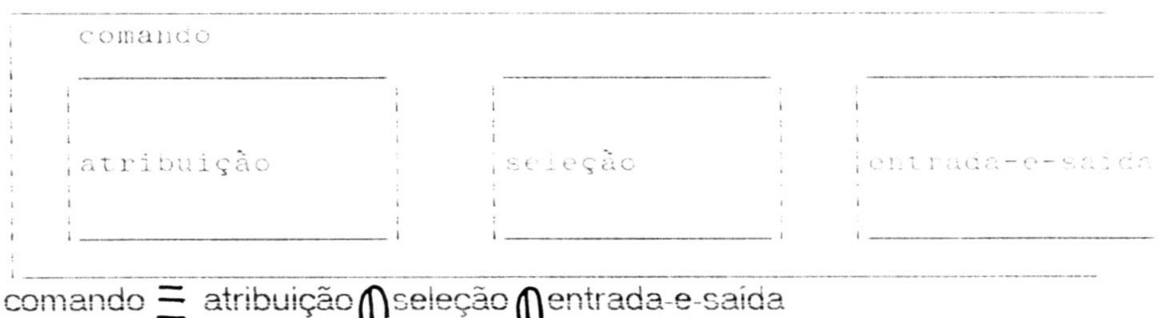
O diagrama de contorno conta com um símbolo básico: o contorno, que pode ser aberto ou fechado. Um contorno aberto representa uma estrutura sintática do ponto de vista abstrato que ainda vai ser refinada no contexto da mesma especificação. Um refinamento se faz com a inserção em um contorno de outros contornos que representam seus componentes. Um contorno fechado representa um objeto sintático que já pode ter uma

formulação concreta ou seja, uma descrição sintática final (em BNF, por exemplo).

Uma especificação começa com um refinamento que representa o nível mais alto (mais externo) da especificação, o que corresponde à raiz de uma árvore, ou ao não-terminal inicial de uma gramática.

Um conjunto de componentes alinhados na vertical indica que tais componentes devem estar presentes, todos, na caracterização desse conjunto. Se o alinhamento for horizontal, deve-se ter apenas uma deles em cada versão do objeto especificado. O segundo caso corresponde à alternativa em BNF. Em nenhum caso se especificam posições relativas de componentes sintáticos entre si: isto compete à sintaxe concreta.

Como exemplo, seja a caracterização da sintaxe abstrata do comando de atribuição como é visto na maioria da linguagens da família ALGOL. Para situar a atribuição em um contexto, é estabelecida primeiramente a regra de que um comando pode ser uma atribuição ou uma seleção ou uma operação de entrada e saída. O diagrama de contorno para isso seria:



comando ≡ atribuição ∩ seleção ∩ entrada-e-saída

A "equação estrutural" acima usa o operador '⋒' para significar que as ideias de atribuição, seleção e entrada-e-saida se interseccionam em comando. Esta intersecção é na verdade uma coincidência: apenas um componente pode (e deve) estar presente numa versão da estrutura. Por sua vez, atribuição se definirá como

atribuição $\equiv$ emissor ⋓ receptor

Ou seja: uma atribuição deve conter dois componentes, sendo um receptor e um emissor. A ordem em que aparecem não é importante: isto deve ser especificado pela sintaxe concreta. O operador '⋓' especifica a união (a composição) de seus operandos para se formar um objeto mais complexo. Neste caso, todos os componentes devem estar presentes.

O contorno fechado representa um componente que só terá definição (um refinamento) numa regra da sintaxe concreta (em BNF, por exemplo). E' o caso do campo receptor, acima, que poderia ser descrito, numa particular sintaxe concreta, como

```
<receptor> ::= <identificador> |
               <identificador> <lista-de-subscritos>
```

Obviamente, em que nível "fechar" uma estrutura que está sendo descrita é uma decisão do especificador, que a tomará em função do grau de liberdade que ele quer deixar para quem for especificar a sintaxe concreta. No exemplo acima, a especificação da sintaxe do componente receptor, por exemplo, fica a cargo de quem o for descrever concretamente. O que se está exigindo é que uma atribuição tenha receptor e emissor. Se se vai adotar o operador 'LET','<-' ou ':=' (ou outro qualquer, ou mesmo nenhum) para ligar estes campos é uma questão da sintaxe concreta, assim como a forma do emissor e do receptor.

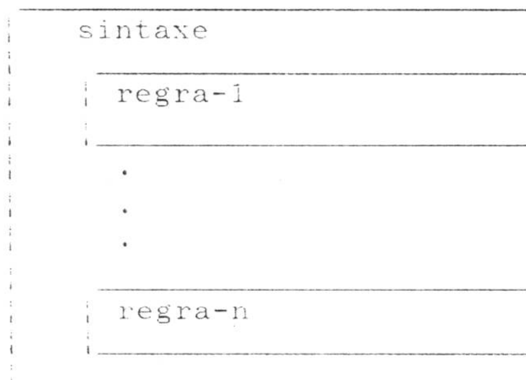
Uma última observação: reticências podem ser usadas para representar a repetição do componente imediatamente anterior um número indefinido de vezes. Pode substituir às vezes a referência recursiva a uma contorno, representando uma iteração.

3.2.2 A Sintaxe Abstrata de SINSEM

Podemos abordar agora a sintaxe abstrata de SINSEM. Em princípio, a especificação de uma linguagem se divide em duas partes: sintaxe e semântica. Da especificação sintática se há de inferir, adicionalmente, a morfologia (o léxico) da linguagem.

$$\text{especificação} \equiv \text{sintaxe} \cup \text{semântica}$$

Os diagramas a seguir completam a especificação.



$$\text{sintaxe} \quad \bigcup_{i=1}^{n} \text{regra-i}$$

A sintaxe é dada por um conjunto de n regras.

Uma regra sintática é um conjunto de declarações, nome da regra e descrição. O componente <u>declarações</u> pode ser vazio, isto é, pode não estar presente. Estas características são representadas pelo diagrama de contorno a seguir.

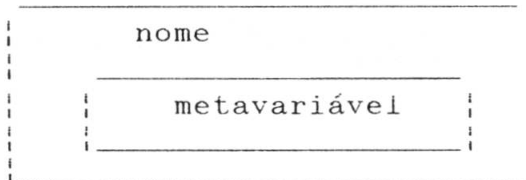
regra $\equiv$

(<vazio> $\cap$ <declarações>)

$\cup$ nome

$\cup$ descrição

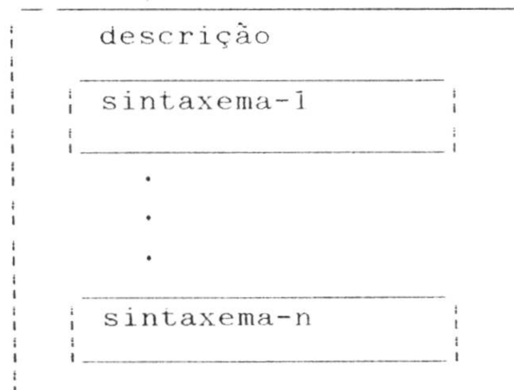
O nome da regra é a metavariável sob definição.



nome $\equiv$

<metavariável>

O seguinte diagrama de contorno representa uma descrição.



descrição

$$\bigcup_{1}^{n} i \quad \text{sintaxema-i}$$

A descrição de um nome (ou seja, de uma metavariável) é um conjunto de "sintaxemas" (unidades básicas da descrição sintática).

O diagrama de sintaxema é o seguinte.

Um sintaxema pode ser um terminal (uma constante sintática), um não-terminal (uma metavariável) ou uma referência semântica (uma forma de associar um significado a uma regra ou sintaxema, como se verá na seção a seguir).

Finalmente, a componente semântica se estabelece como um conjunto de "semas" (unidades básicas de descrição semântica).

Um sema, dada a abordagem construtiva interpretativa (cf. seção 3.1), é uma rotina escrita em alguma linguagem de programação ou de descrição de algoritmos.

3.3 A Metalinguagem SINSEM

Nesta seção será discutida a linguagem de descrição de linguagens do sistema SINSEM, que, por sinédoque, também se chama 'SINSEM'.

3.3.1 Introdução

SINSEM é um sistema de produções construído a partir da notação BNF, com extensões para exprimir a semântica de uma linguagem, de um modo construtivo.

De um ponto de vista formal, uma especificação E se constitui de uma gramática G e de um conjunto S de rotinas semânticas, ou seja,

$$E \equiv G \uplus S$$

Uma gramática em SINSEM é uma 5-upla

G = (Vn, Vt, Vs, P, Z),

onde Vn é o conjunto de não-terminais;

Vt é o conjunto de terminais;

Vs é o conjunto de referências semânticas;

P é o conjunto de produções;

Z é o não-terminal inicial.

A seguir são melhor explanados esses termos. Não-terminal é um identificador usado para nomear uma produção, isto é, trata-se de uma metavariável. Terminal é qualquer cadeia de caracteres elementares da linguagem sob definição, ou seja, uma constante sintática da linguagem de uso. Produção é uma regra de substituição de um não-terminal por uma forma sentencial que lhe corresponda, isto é, por uma sua descrição, dada por uma seqüencia de terminais, não-terminais e referências semânticas. Referência semântica é um identificador que nomeia uma rotina semântica a ser executada num dado ponto de aplicação de uma produção. As referências semânticas comumente tem argumentos. Rotina semântica (ou sema) é um procedimento (comumente parametrizado) a ser executado quando, durante uma aplicação (isto é, substituição, expansão) de um não-terminal, em um processo de análise sintática descendente

derivado dessa gramática, for detectada uma referência semântica correspondente. Não-terminal inicial é o identificador da produção (única) através da qual deve ser iniciado o processo de análise sintática (em SINSEM, é sempre a primeira da gramática).

A especificação é dita construtiva porque dado um programa

$$P = \mathrm{CAT}_{i=1}^{n} \text{ string-i}$$

a tradução de P é

$$T(P) = \mathrm{CAT}_{i=1}^{n} \; S[\, c(i)\,](\text{ string-i })$$

ou seja, dado um programa, que é a concatenação de uma seqüência de n "strings", sua tradução é a concatenação, na mesma ordem, dos resultados da aplicação de uma função tradutora S[c(i)] sobre cada string-i. A família de funções S é que incorpora a semântica da linguagem. A função c:N --> N é a "função de escolha", que invoca, para um dado string-i, a correspondente função tradutora S[c(i)]. O tipo da família S é

S:(N --> N) --> (string --> string)

e sendo string-k a tradução de string-i, segundo a função-membro de S escolhida pela função c:N --> N, uma outra expressão válida é

S[c(i)](string-i) = string-k

Num sistema como SINSEM, as funções de tradução recebem nomes e não índices. O argumento dado à função c é o nome da rotina semântica que aparece numa referência semântica numa regra. A família de funções de tradução se reduz ao conjunto S de rotinas semânticas. A função de escolha c é provida pelo sistema como parte do tradutor a construir, e, dada uma referência semântica, que é o meio que o especificador tem de indexar uma aplicação da família de funções de tradução, ele, o tradutor, a converte numa invocação a uma rotina semântica.

3.3.2 Sintaxe Concreta e Semântica Informal de SINSEM

Neste item se descreverá a linguagem de especificação SINSEM. A sintaxe será expressa em BNF (com a extensão de que construções entre colchetes sao opcionais). A semântica de cada construção será comunicada informalmente.

Como o formalismo sintático subjacente é BNF, as gramáticas aceitas são as livres do contexto, com a seguinte RESTRIÇÃO: não são permitidas produções com recursões 'a esquerda, visto que a gramática vai dar origem a um reconhecedor sintático recursivo descendente.

a) Forma geral de uma especificação

Uma especificação de uma linguagem consta de duas partes: sintaxe e semântica. Em SINSEM a sintaxe é dada em BNF. A semântica de cada construção sintática é associada a ela de forma abreviada no corpo da própria descrição sintática, e depois detalhada na forma de procedimentos na parte da descrição semântica, que é o conjunto de rotinas semânticas. A seguir é dada a sintaxe concreta de uma especificação.

---

1. <especificação> ->

   LINGUAGEM <identificador>.

   BIBLIOTECA <identificador>.

   [VARTRAB <declarações>.]

   SINTAXE <lista-de-regras>.

   SEMÂNTICA <lista-de-semas>.

Primeiramente, se identifica a linguagem sob definição: é pelo identificador que a nomeia na declaração LINGUAGEM que o seu tradutor, resultante do processamento desta especificação, será invocado. Em seguida, se indica, na declaração BIBLIOTECA, o arquivo que contém um conjunto de primitivas sintatico-semânticas, construções pré-definidas, o qual deve se agregar `a presente especificação. Há de ser útil ao usuário nao precisar descrever construções corriqueiras como identificadores,

constantes inteiras, cadeias de caracteres, etc., ou mesmo alguma outra, mais particular, mas que seja frequente nas suas definições (do usuário). Através do envelope, o usuário pode estabelecer a sua biblioteca sintatico-semântica; senão, poderá referir-se ao arquivo STANDARD, provido pelo sistema, e que será apresentado na seção 4.2.

A declaração VARTRAB (opcional) serve para se declarar um conjunto de <u>variáveis de trabalho</u> que serão usadas para se registrar, ao longo de toda a especificação, dados referentes ao processo. Tais variáveis serão manipuladas pelas rotinas semânticas `a medida que forem sendo invocadas pelas referências semânticas. Variáveis de trabalho também podem declarar-se localmente a cada regra sintática, mas aquelas declaradas neste nível agora em discussão são globais, isto é, são conhecidas em todas as regras. Os tipos possíveis para as variáveis de trabalho são discutidos na alínea b, subseqüente.

A declaração SINTAXE introduz a gramática, considerada já no item 3.3.1, e que se constitui simplesmente numa seqüência de regras sintáticas, isto é, de produções, em tudo semelhantes `aquelas de BNF, apenas com extensões mínimas com vistas a exprimir abreviadamente a semântica da linguagem de uso, a qual será dada, então, de uma forma construtiva interpretativa por associação de significados aos terminais ou não-terminais (como foi caracterizado na seção 3.1).

Por fim, a declaração SEMANTICA encabeça o conjunto de procedimentos que modelam a semântica da linguagem de uso, ou seja ,

os semas, ou rotinas semânticas, associadas `as referências semânticas que fazem parte das produções.

b) Variáveis de trabalho

Para acompanhar ou mesmo controlar numa certa medida o processo de tradução, é possivel se manter registro de dados ao longo desse processo através de variáveis que são manipuladas pelas rotinas semânticas. Elas são ditas variáveis de trabalho. Como cada produção vai dar origem a uma rotina do analisador sintático a ser produzido, pode-se declarar variáveis locais a cada produção. Isto se faz através da declaração VARTRAB, cuja sintaxe é dada a seguir.

---

2. <declara-var-trab> ->

   VARTRAB <declarações>.

3. <declarações> ->

   <declaração> I

   <declaração> ; <declarações>

4. <declaração> ->

   <identificador> : <tipo>

5. <tipo> ->

   INTEIRO I STRING I PILHA-INT I PILHA-STR

UFRGS
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

O tipo INTEIRO qualifica uma variável a armazenar valores inteiros.

STRING qualifica uma variável como uma cadeia de caracteres variável de tamanho máximo igual a 80.

PILHA-INT descreve uma pilha de inteiros e PILHA-STR uma pilha de "strings".

E' bom repetir que as pilhas citadas são "pilhas-vetores": são operadas como pilhas, mas podem ter seus elementos inferiores acessados por duas operações tipicas de vetores: PESQ e ALTERA (cfe. a seção 3.3.3).

As operações sobre inteiros e cadeias de caracteres são aquelas propiciadas pela linguagem de trabalho. mas só são providas ao especificador através das rotinas semânticas que ele mesmo estabeleça (em L-trab). Para as pilhas há um conjunto de rotinas semânticas padrão que são fornecidas pela biblioteca STANDARD. Estas rotinas padrão são discutidas na seção 3.3.3 e no capítulo 4.

Exemplos de declarações de variáveis estão junto `a próxima alinea.

c) Produções

As produções (ou regras sintáticas) vão estabelecer a sintaxe da linguagem de uso e associá-la com a sua semântica. Uma regra sintática tem a seguinte forma geral.

---

6. <regra> ->

   REGRA <inteiro> :

   [declara-var-trab>]

   <metavariável> -> <lista-de-alternativas> FIM

7. <lista-de-alternativas> ->

   <alternativa> |

   <alternativa> OU <lista-de-alternativas>

8. <alternativa> ->

   <sintaxema> | <sintaxema> <alternativa>

9. <sintaxema> ->

   '<string>' | <metavariável> | <ref-semântica>

Vê-se pelas regras BNF acima que uma produção é introduzida pela palavra REGRA, é numerada, e é nomeada por uma metavariável, isto é, por um não-terminal que poderá ser referido em outra produção. Cada não-terminal, como sói ser em BNF, pode ser substituído por mais de uma forma sentencial, que aqui sao chamadas alternativas. Cada alternativa, por sua vez, é uma lista de sintaxemas, ou seja, de unidades básicas de descrição sintática. Um sintaxema pode ser:

- um "string" entre apóstrofes, o que será interpretado como um terminal da linguagem de uso (tal "string" deve conter ao menos um caracter, isto é, nao pode ser vazio: para tanto existe a metavariável pré-definida <vazio>, que trata adequadamente este caso);

- uma metavariável, isto é, um não-terminal, que invoca a aplicação de outra produção assim nomeada;

- uma referência semântica, que invoca uma rotina semântica correspondente ao significado da construção cuja análise sintática tenha sido concluída com sucesso (terminal ou não-terminal imediatamente anterior).

---

10. <metavariável> ->

    ≤ <identificador> ≥

11. <ref-semântica> ->

    $<identificador> [(<lista-de-argumentos>)]

12. <argumento> ->

    <inteiro> | '<string>' | <identificador> | <metavariável>

13. <lista-de-argumentos> ->

    <argumento> [, <lista-de-argumentos>]

As metavariáveis são identificadores entre parênteses angulares. Uma referência semântica é um identificador, precedido de um '$', seguido ou não de argumentos. Os argumentos podem ser constantes, do tipo INTEIRO ou STRING, variáveis ou metavariáveis. No caso de ser uma metavariável, seu conteúdo é o "substring" do programa-fonte em linguagem de uso que por último tenha estado em correspondência com tal metavariável, no contexto da mesma alternativa, durante a análise sintática em curso (pensando sempre do ponto de vista do programa tradutor gerado).

Junto a uma regra pode-se declarar variáveis de trabalho locais. Elas tambem serão herdadas pelas regras que vierem a ser citadas (pela sua metavariável) nas formas sentenciais (alternativas) desta regra

Ou seja, cada regra estabelece um "ambiente" em termos do alcance de suas variáveis de trabalho, ao estilo ALGOL: uma sub-rotina (regra) compartilha os dados declarados no contexto da sua invocante (a menos que sejam redeclarados, com o mesmo nome).

Um exemplo de produção é dado a seguir:

```
REGRA 10:
<bloco> -> <decl-var><decl-proc> $GEN3('BLK',BN,LENGTH)
                'BEGIN' <lista-com> 'END' $GEN1('BEX')
FIM
```

Ou seja: uma construção <bloco> é uma construção <decl-var>, seguida de uma construção <decl-proc>, seguida de 'BEGIN', seguida de uma construção <lista-com> seguida de 'END'. Adicionalmente, se especifica que o significado de <decl-proc>, nesse contexto, é dado pela rotina GEN3 (isto é, GEN3 será executada se a construção <decl-proc> se confirmar) e que o significado de 'END', nesse contexto, é dado pela rotina GEN1 (isto é, GEN1 será executada se 'END' for encontrado). Ambas as rotinas semânticas são particularizadas por seus argumentos, isto é, "significam" um conjunto de "sentidos" em geral, dentre os quais algum deles é fixado pelos argumentos fornecidos neste ponto.

Um outro exemplo, envolvendo declaração de variáveis de trabalho e com mais de uma alternativa, é o seguinte:

REGRA 2: VARTRAB

L1: INTEIRO.

<rest-if> -> 'END IF' $ASSIGNLBL(L)

OU 'ELSE' $NEWLBL(L1)

$GEN2('JMP', L1)

$ASSIGNLBL(L)

<stmt-list> 'END IF' $ASSIGNLBL(L1)

FIM

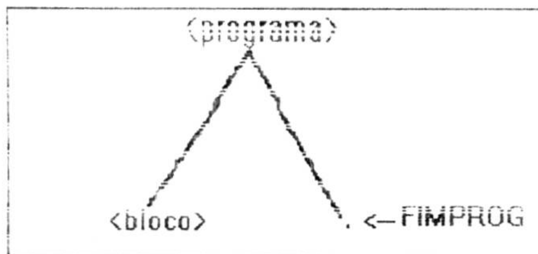
Neste caso, nota-se que o significado atribuído ao terminal 'ELSE' é composto por uma seqüência de três referências semânticas, na segunda alternativa.

d) Significado de uma produção

Os termos semânticos do formalismo constituem uma terceira classe , junto com os terminais e não-terminais da BNF subjacente.

Os termos semânticos são ditos referências semânticas e são distinguidos por serem precedidos do caracter '$'. Equivalem a chamadas de rotinas a serem executadas quando do reconhecimento de uma construção, representada pela metavariável ou terminal que a antecede imediatamente , na regra.

Uma regra pode ser representada por uma árvore. Como exemplo, a figura 3.1 é a árvore de uma regra, onde se observa que:



REGRA 5: <programa> -> <bloco> '.' $FIMPROG FIM

Fig. 3.1 Árvore de uma regra

- uma construção <programa> é uma construção <bloco> seguida de um ponto ('.'):

- o significado "armazenado" na rotina FIMPROG é atribuído ao ponto, sempre que este fizer parte de uma construção <programa>, de acordo com a sintaxe especificada, isto é, quando se seguir a uma construção <bloco>.

Há condições de se atribuir significados a qualquer elemento terminal ou não-terminal de uma regra, como já se exemplificou na alínea c, acima.

Outra associação possível é a de um significado (isto é, uma referência semântica) com a metavariável sob definição, ou seja com a raiz da árvore. Isto ocorre ao se inserir, numa alternativa qualquer de uma regra, uma referência semântica como primeiro sintaxema. Por exemplo:

REGRA 130: <isto> -> $AQUILO <aquele-outro> ' ' FIM

Neste caso, o significado AQUILO é atribuido `a raiz da regra, ou seja `a metavariavel <isto>, que, sempre que for citada em outra construção,implicará na execução da rotina AQUILO como primeira atividade durante a tradução. Uma forma de representar isso é a da figura 3.2. Note-se a atribuição `a raiz graficada `a esquerda do nodo.

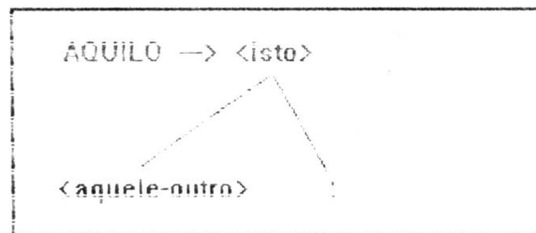


Fig. 3.2 Árvore de uma regra com significado atribuido `a raiz.

Em função das considerações acima, pode-se cogitar do <u>significado de uma produção</u>. Verifica-se que o significado de uma produção é a lista dos significados de seus componentes segundo o caminhamento pré-fixado.

Dito de outra forma, o significado de uma produção é a concatenação dos significados de seus componentes, sendo que

- o significado de um terminal é ou nulo ou o significado dado pela referência semântica que o segue;

- o significado de um **não-terminal** é o significado obtido pela sua substituição concatenado com o significado que lhe é atribuido na própria produção.

Antes de se apresentar um exemplo, há que se frisar que o significado atribuido a um não-terminal **não é** o significado da raiz da sua árvore, uma vez que este é independente do contexto onde aparece, enquanto aquele é atribuido ao não-terminal somente na produção onde foi citado. Para deixar mais clara essa diferença, seja a metavariável <isto> do exemplo anterior, agora componente de uma construção descrita na produção a seguir.

```
REGRA 140:
    <nisto> -> <em> <isto> $CONTRAÇÃO
FIM
```

A figura 3.3 procura representar o significado da regra <nisto> detalhando o que se sabe da regra <isto>. Ela comunica que o significado

CONTRAÇÃO só se concatena `a lista de significados de <nisto> depois da análise deixar a regra <isto>.

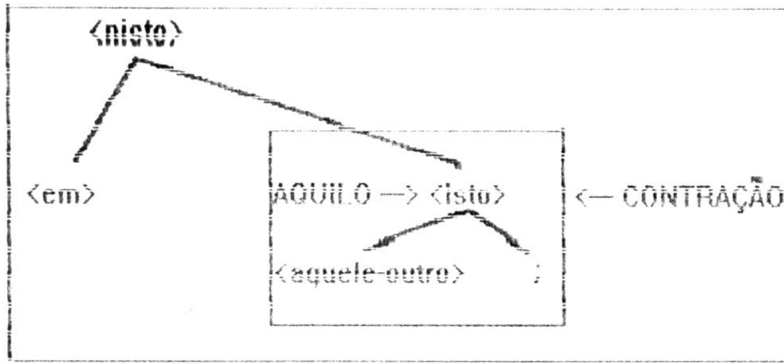


Fig. 3.3 Significado atribuido a um não-terminal cuja produção correspondente tem um significado atribuido `a raiz.

Para simplificar a graficação das árvores sintatico-semânticas é que se nota o significado da raiz `a esquerda do nodo. Significados de não-terminais enquanto componentes ficam `a direita do nodo.

Com base no que foi colocado acima, pode-se estabelecer o significado de uma produção, considerando a definição a seguir.

```
REGRA 20:
    <var0> -> 'ITEM1' <var1> $X 'FIM' $Y
    OU        'ITEM2' <var2> 'FIM' $Y
FIM
```

REGRA 21:

<var1> -> $Z 'TIPO' <um> $ROT1 <dois>

FIM

Uma árvore possível, conforme uma opção feita na regra 20, é dada na figura 3.4. Com base nela, e obedecendo o caminhamento pré-fixado, listam-se os significados componentes da regra 20 nesse caso, usando a função

sig(x) significado da metavariável x.

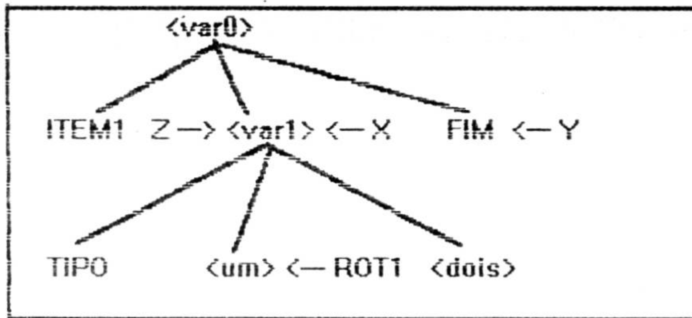


Fig 3.4 Árvore sintatico-semântica da produção descrita no texto(REGRA 20).

sig(<var0>) sig(<var1>) || X || Y

Como o significado de <var1> é dado por

sig(<var1>) Z || sig(<um>) || ROT1 || sig(<dois>)

tem-se que

sig(<var0>) Z || sig(<um>) || ROT1 || sig(<dois>) || X || Y

E' interessante observar que, se as referências semânticas estabelecem a execução de rotinas de geração de código, ter-se-á, em linhas gerais, uma relação linear entre o andamento da análise sintática descendente e o "string" de código objeto obtido. Na prática esta relação é perturbada pela necessidade de, muitas vezes, postergar uma geração de código até que se verifique uma dada condição sintática.

e) Rotinas semânticas

Conforme se mostrou na alínea a desta seção, a semântica da linguagem de uso é dada na forma de uma lista de "semas", ou seja, de rotinas semânticas, que se relacionam com a gramática dada na especificação da sintaxe através do que se denomina referências semânticas. Com o intuito de completar a especificação BNF de SINSEM é dada a seguir a sintaxe de um sema.

---

14. <sema> -> SEMA [(<lista-de-parâmetros>)] :

    <lista-de-comandos>

    FIM

15. <lista-de-parâmetros> -> parâmetro |

    <parâmetro> ; <lista-de-parâmetros>

16. <parâmetro> -> <declaração> |

    <identificador>:METAVAR

A <lista-de-comandos> representa o corpo de um procedimento sub-rotina escrito na linguagem de trabalho. No nível em que presentemente se encontra esta proposta, a linguagem de trabalho se restringe a ser a linguagem de implementação do próprio sistema: Pascal.

Os argumentos são passados por referência, por uma questão de simplicidade. Ao ser transformada em Pascal, a lista de parâmetros é simplesmente antecedida de VAR).

Observe-se que se adiciona um tipo `aqueles possíveis para variáveis de trabalho: é o tipo METAVAR, que especifica como argumento uma metavariável, o que se traduzirá num par de inteiros, apontadores para o início e o fim do texto correspondente (no programa-fonte) ao trecho abrangido pela metavariável argumento.

Um exemplo de sema é dado a seguir.

```
SEMA ASSIGN ( RECEPTOR:INTEIRO, EMISSOR:INTEIRO ) :
  BEGIN
    RECEPTOR := EMISSOR
  END;
FIM
```

Uma referência semântica do tipo

$ASSIGN(KONT , 0)

será usada para atribuir 0 `a variavel de trabalho KONT.

3.3.3 Semântica Estática em SINSEM

O mesmo mecanismo usado para especificar a semântica (dinâmica) pode ser usado para se especificar a semântica estática (ou as regras "sensíveis ao contexto"), através de um subconjunto das rotinas semânticas do tipo

Se : string ---> $\emptyset$

significando que não retornam resultados na forma de "strings" de tradução, mas apenas têm "efeitos colaterais" sobre o processo de tradução, quais sejam os efeitos de verificação da semântica estática.

O especificador pode escrever suas próprias rotinas semânticas, mas, para auxiliá-lo na tarefa da especificação da semântica estática especialmente, o sistema fornece um conjunto de semas pré-definidos como "significados estáticos padrão". Estas rotinas foram escolhidas dentre um conjunto proposto, com a mesma finalidade, por /WIL80/. A forma de invocação é dada a seguir. Note-se que, como de

resto qualquer rotina, elas podem ser invocadas como referências semânticas ou como invocações normais no corpo de outros semas. Note-se tambem que os tipos de alguns parâmetros são generalizados, isto é, podem receber argumentos de mais de um tipo, bastando para tanto que seja mantida a coerência com os demais argumentos.

A lista de rotinas padrão para a semântica estática é a seguinte.

- PUSH(P:pilha, V:tipo-da-pilha) - empilha V em P podendo P ser do tipo PILHA-INT ou PILHA-STR, devendo então V ser do tipo INTEIRO ou STRING respectivamente.

- POP(P:pilha) - desempilha o elemento no topo de P.

- TOP(P:pilha , V:tipo-da-pilha) - faz a consulta a P, isto é, coloca em V o conteúdo do topo de P. Valem as observações feitas para PUSH.

- POPTOTAL(P:pilha) - esvazia P.

- PESQ(P:pilha, V:tipo-da-pilha, I:INTEIRO) - pesquisa o valor V na pilha P e coloca seu índice em I. Se o valor V não é encontrado em P, I recebe 0. Este índice I poderá ser usado subsequentemente por outras rotinas, como COPIA e ALTERA, por exemplo.

- COPIA(P:pilha. I:INTEIRO. V:tipo-da-pilha) - faz a cópia para a variável V do conteúdo de P indexado por I.

- ALTERA(P:pilha. I:INTEIRO. V:tipo-da-pilha) - altera a posição de P indexada por I para passar a conter o valor da variável V.

- PROBLEMA - estabelece a marca de insucesso como resultante da execução de uma rotina semântica (normalmente o retorno de uma ref-semântica sempre é marcado como sucesso; caso ela invoque PROBLEMA. as referências semânticas que a seguem não serão mais executadas. exceto ERRO).

- ERRO(N:INTEIRO) - esta rotina especifica a "memorização" por parte do tradutor do número de uma mensagem de erro a ser emitida apos a listagem do programa-fonte. junto ao número de ordem do registro corrente do programa-fonte. ou seja. o "número da linha" onde ocorreu o erro. Uma mensagem pode ser associada a este número N. argumento da rotina ERRO. através do comando ERROS do envelope (conforme a seção 4.1). O que ERRO tem de mais particular é que. contrariamente `as demais rotinas semânticas. tem sua execução marcada somente para quando o sintaxema que a antecede tenha resultado numa hipótese fracassada (pode ser vista como o "anti-significado" padrão).

Um exemplo de especificação de semântica estática é dado a seguir. Trata-se da regra de que toda variável deve ser declarada. A verificação se faz. no exemplo. sobre um comando de atribuição. mais especificamente sobre o campo receptor.

REGRA 9: <decl-var> ->

'VARS' <lista-var> ';' FIM

REGRA 10: <lista-var> ->

<identificador> $PUSH(VARS.<identificador>)

OU <identificador> $PUSH(VARS.<identificador>)

',' <lista-var>

FIM

...

REGRA 50:<atribuição> ->

<identificador> $VERIF(VARS. <identificador>)

$ERRO(5)

':=' <expressão>

FIM

A regra 10 diz que para cada identificador na lista de variáveis da declaração VARS (definida na regra 9), deve haver uma inserção correspondente na pilha VARS. Na regra 50, por outro lado, se invoca uma rotina VERIF, significado do nãoterminal <identificador> nesta regra, que deverá checar a presença ou não do mesmo na pilha de variáveis. Caso não o encontre, acionará a rotina PROBLEMA para marcar o erro encontrado. O sema VERIF é dado a seguir.

```
SEMA VERIF( X:PILHA-STR. Y:METAVAR );
 VAR IND:INTEGER;
 BEGIN
  PESQ(X. Y. IND );
  IF IND = 0 THEN PROBLEMA
 END;
FIM
```

Observe-se o corpo da rotina semântica. Ele está em Pascal. com o que o especificador é o responsável pela sua correção. Uma vez passado o cabeçalho. o sistema SINSEM só se preocupa em ajustar internamente as variáveis da lista de parâmetros. estando também atento `a palavra FIM. motivo pelo qual é reservado seu uso para concluir a rotina semântica.

3.3.4 Recuperação de Erros em SINSEM

Embora a recuperação de erros sintáticos seja uma atividade de nível mais baixo que a especificação da sintaxe. no sentido de que está intrinsecamente relacionada com a forma de implementação do tradutor. optou-se aqui por dar ao usuário uma condição mínima de especificação da recuperação de erros.

No tradutor gerado por SINSEM a recuperação de erros se faz pelo processo conhecido como "panic mode" /HAM84/. Este processo consiste no seguinte procedimento: ao ser constatada uma situação de desacordo entre um terminal obtido e o terminal esperado, dado pela regra sintática que está sendo usada como hipótese de análise, ignora-se o texto subsequente até se encontrar um terminal que pertença a um conjunto dito <u>de sincronização</u>, a partir do qual a análise é retomada. O terminal de sincronização não precisa pertencer necessariamente `a regra adotada como hipótese.

O sistema SINSEM adota como conjunto de sincronização, para cada regra, o conjunto de terminais a ela pertencentes.

O usuário pode no entanto, especificar um conjunto de sincronização diverso, através da rotina semântica SINCRON, cuja forma geral de invocação é

```
SINCRON( '<lista-de-terminais>' )
```

A partir da execução de uma referência a SINCRON a sua lista de terminais é que se torna o conjunto de sincronização vigente. Um exemplo é dado a seguir.

```
REGRA 15: <chamada> ->
            'EXEC' $COD( 'EX' ) $SINCRON( '( . ;')
            <identificador> <args> ';'
FIM
```

REGRA 16: <args> ->

'(' <arg-list> ')'

FIM

Na regra 15 se estabelecem os caracteres '(' e ';' (abre-parênteses e ponto-e-virgula) como sincronizadores: caso 'EXEC' seja reconhecido, ele saira do conjunto de sincronização, devido ao efeito do seu significado SINCRON. Note-se que SINCRON não interrompe a análise, mas apenas muda o conjunto de sincronização. Se 'EXEC' não for reconhecido, o conjunto de sincronização seguirá sendo ['EXEC', ';'], uma vez que SINCRON não será executada.

Caso haja erro, a sincronização em SINSEM deixa o cursor do analisador sintático posicionado no caracter imediatamente anterior ao terminal sincronizador, de modo a permitir o andamento normal da análise sintática.

3.4 A construção do Tradutor

O tradutor produzido pelo sistema SINSEM é um analisador sintático recursivo descendente dotado de ações semânticas. Cada regra em SINSEM dará origem a uma rotina de análise sintática, as quais se agregarão ao conjunto de rotinas semânticas fornecidas pelo especificador e pela biblioteca sintatico-semântica.

A gramática G é vista como uma matriz de m linhas de tamanho variável. Cada linha é uma regra. Cada elemento x(i,j) da linha-i é um sintaxema.

Um algoritmo básico de construção seria o seguinte.

```
CONSTRUIR UM TRADUTOR.
leia uma regra;
enquanto não terminar a gramática faça
    construa uma rotina sintática correspondente;
    leia uma regra
fim enquanto
leia uma rotina semântica;
enquanto não terminarem  as rotinas semânticas faça
    adapte a rotina semântica;
    leia uma rotina semântica
fim enquanto
FIM
```

No algoritmo acima, "adaptar uma rotina semântica" quer dizer torná-la compatível com a linguagem de trabalho adotada (veja a seção 2.1).

A construção de uma rotina sintática a partir de uma regra é esboçada pelo próximo algoritmo.

CONSTRUIR UMA ROTINA SINTÁTICA.

construa uma cabeçalho;

para cada sintaxema da regra faça

caso o sintaxema seja

terminal: construa um "matching" para ele;

não terminal,

ref-semântica: construa uma chamada `a rotina

correspondente;

outro: erro

fim caso

fim para

FIM

Construir um "matching" (isto é, um "emparelhamento", ou "comparação") quer dizer estabelecer um acesso ao programa-fonte para obter o próximo terminal e compará-lo com o terminal especificado nessa posição sintática pela regra sendo processada.

As rotinas sintáticas ganham todas um parâmetro lógico que vai assinalar o sucesso ou fracasso da análise sintática empreendida. Este parâmetro é usado pelo sistema para marcar a execução ou não dos procedimentos de recuperação de erros e das rotinas semânticas. Estas só são executadas se o sintaxema a que se ligam tenha sido marcado com 'sucesso' (a exceção é a rotina ERRO, que só é invocada se o sintaxema anterior tenha sido marcado com 'fracasso').

As rotinas semânticas também tem esse parâmetro adicional. Normalmente o sistema o seta em 'sucesso', mas o usuário pode explicitamente marcá-lo como fracasso através da rotina PROBLEMA.

4 O AMBIENTE DE IMPLEMENTAÇÃO

Para auxiliar o especificador no uso do sistema SINSEM, é-lhe oferecido:

a) um programa que se constitui numa interface entre este e aquele;

b) um arquivo de regras e rotinas semânticas predefinidas.

4.1 O Envelope

O programa chamado "envelope" é o responsável pela parte interativa do sistema. Através dele é que são criadas e geridas as "sessões de definição". Além de controlar implicitamente a sessão, ele ainda obedece a alguns comandos explícitos do usuário. A parte implícita se encarrega de abrir e fechar as sessões, executando as inicializações necessárias e também vai formando telas correspondentes às atividades que vão sendo desenvolvidas.

O envelope estabelece para cada usuário um arquivo de trabalho identificado com este usuário. Neste arquivo são armazenadas as regras que definem uma linguagem e as mensagens de erro que serão usadas pelo seu tradutor.

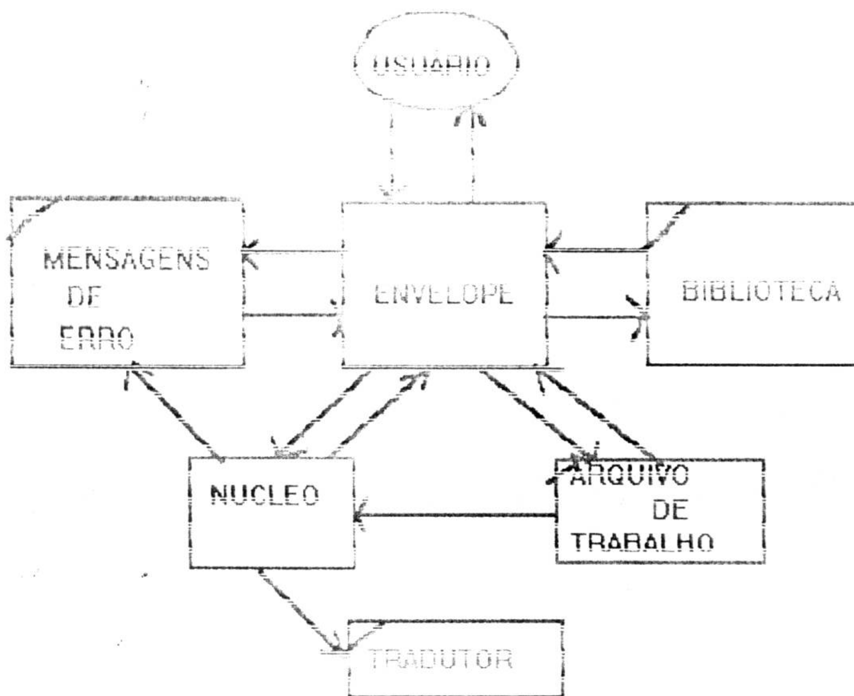


Fig. 4.1 Diagrama de Sistema do Sistema SINSEM

Adicionalmente, liga-se ao usuário um arquivo para conter o tradutor, resultado do trabalho de construção do mesmo a partir das regras de especificação.

A figura 4.1 representa estas relações. O usuário ativa o envelope para construir sua especificação, que será armazenada no arquivo de trabalho. O envelope, sob uma ordem do usuário, ativa o núcleo SINSEM, que tomará o arquivo de trabalho e tentará produzir um

tradutor com base no seu conteúdo. Alternativamente, o núcleo pode comunicar-se com o envelope e este com o usuário, de modo a se poder verificar o andamento do processo, por exemplo, listando o arquivo de mensagens de erro que o núcleo estabelece ao processar o arquivo de trabalho.

O envelope também se responsabiliza pelo acesso `a biblioteca SINSEM, sob controle do usuário ou do núcleo do sistema.

Os comandos que podem ser dirigidos ao envelope são os seguintes (conforme a tabela 4.1).

O comando CRIA estabelece um arquivo de trabalho para o usuário que se nomeia com <identificador>. Se já existe esse arquivo, apenas o conecta `a sessao. A opção BIBLIO especifica que este arquivo será uma biblioteca sintatico-semantica.

O comando ENTRA faz com que o envelope passe a armazenar no arquivo de trabalho as regras sintáticas e rotinas semânticas que o usuário vai fornecendo, até encontrar um asterisco ('*') na primeira coluna da linha de entrada, depois de uma palavra chave 'FIM' ou 'FIM.', que encerra uma regra. Uma regra dada com o mesmo número que uma já éxistente, substitui esta última. Uma regra com mesma

metavariável que outra já existente, entra como uma nova alternativa para a mesma regra.

TABELA 4.1 Lista de Comandos ao Envelope do sistema SINSEM *

---

1. CRIA <identificador> [BIBLIO]
2. ENTRA
3. CONSTROI
4. FIM
5. LIMPA
6. LISTA [<metavar> | ESPECIF | ERROS | TRAD | MENS | BIBLIO]
7. GUIA ENTRA
8. GUIA CONSTROI
9. ENTRA CONSTROI
10. GUIA ENTRA CONSTROI
11. TCHAU
12. ERROS
13. EDITA
14. INCLUI
15. EXCLUI <metavar>

O comando ENTRA faz com que o envelope passe a armazenar no arquivo de trabalho as regras sintáticas e rotinas semânticas que o usuário vai fornecendo, até encontrar um asterisco ('*') na primeira coluna da linha de entrada, depois de uma palavra chave 'FIM' ou 'FIM.', que encerra uma regra. Uma regra dada com o mesmo número que uma já existente, substitui esta última. Uma regra com mesma metavariável que outra já existente, entra como uma nova alternativa para a mesma regra.

O comando CONSTROI faz o envelope ativar o núcleo do SINSEM para que este tome o arquivo de trabalho e produza o tradutor especificado. Caso haja erros na especificação, mensagens são armazenadas no arquivo de mensagens de erro. Ao ser reativado o envelope, este informa o usuário da existência dessas mensagens, e elas poderão ser listadas na tela.

O comando FIM fecha os arquivos e os desconecta da sessão de definição, sem encerrar esta última.

O comando LIMPA esvazia o arquivo de trabalho, isto é, deleta o seu conteúdo.

O comando LISTA solicita a listagem do arquivo de trabalho, quando desacompanhado das suas opções, que são as seguintes (exclusivas entre si), com seus respectivos efeitos:

- <metavar> - será listada apenas a regra nomeada com <metavar>;

- ESPECIF - será listada apenas a especificação até o momento armazenada, ou seja, o conjunto de regras;

- ERROS - serão listadas apenas as mensagens de erro dadas pelo usuário (através do comando ERROS) para serem conectadas no tradutor às referencias semanticas ERRO(<n>) presentes na especificação;

- TRAD - será listado o tradutor resultante (o texto existente até o momento);

- MENS - será listado o arquivo de mensagens produzidas pelo núcleo durante a última execução do comando CONSTROI;

- BIBLIO - será listada a biblioteca.

O conjunto GUIA ENTRA tem o efeito de ENTRA, mas a entrada vai sendo formatada pelo envelope.

O conjunto GUIA CONSTROI funciona como CONSTROI, mas caso haja erros, guia o definidor na correção, no que for possível.

O conjunto ENTRA CONSTROI funciona como ENTRA, mas a cada regra completada, o sistema tenta já processá-la, de modo a se adicionar a rotina correspondente ao arquivo do tradutor.

O conjunto GUIA ENTRA CONSTROI equivale a uma combinação GUIA ENTRA com ENTRA CONSTROI, isto é, a cada regra completada, processa-a, e, adicionalmente, se houver erros, guia o especificador numa possível correção.

O comando TCHAU encerra a sessão de definição, deletando também o arquivo de mensagens de erro.

O comando ERROS dá condições ao usuário de estabelecer uma lista de mensagens de erro, numeradas, que poderão ser referidas

posteriormente, durante a especificação, pela rotina ERRO (veja a seção 3.3.3). Depois de ser dado o comando ERROS, o envelope passa a armazenar no arquivo de trabalho as mensagens, fornecidas no formato

<número> <string>

onde <número> é um inteiro de 1 a 999 e <string> é um texto de até 80 caracteres, terminando em ponto ('.'). Ao ser dada entrada a um <número> igual a '000' o comando ERROS é encerrado. Números repetidos simplesmente causam a substituição da respectiva mensagem.

O comando EDITA invoca um editor de textos existente na instalação, dando condições ao usuário de alterar diretamente o arquivo de trabalho.

Os comandos INCLUI e EXCLUI são tratados na seção a seguir.

4.2 A Biblioteca Sintatico-Semântica

SINSEM fornece ao especificador algumas estruturas sintáticas pré-definidas, bem como algumas rotinas semânticas. Outras podem ser agregadas pelo usuário através do envelope.

As estruturas sintáticas pré-definidas sao:

- <identificador> - um identificador tradicional: 12 caracteres alfanuméricos, começando por letra, podendo conter hifens;

- <inteiro> - um inteiro sem sinal;

- <car-especial> - um caracter especial existente na instalação;

- <vazio> - o "string" vazio;

- <string> - um "string" de caracteres, entre aspas simples.

As rotinas semânticas pré-definidas são aquelas abordadas no item 3.3.3, que visam o tratamento da semantica estatica, sem se descartar sua utilidade para outros usos, e ainda as seguintes, que se relacionam mais com a especificação da semântica concreta:

- COD(S:STRING) - agrega ao "string-objeto" o string S, ou seja, grava-o no arquivo nomeado pelo definidor na declaração LINGUAGEM (veja a seção 3.3.2, alínea a);

- INTCHAR(I:INTEIRO, S:STRING) - converte o inteiro I num "string" de dígitos decimais e o armazena em S;

- CHARINT(S:STRING, I:INTEIRO) - converte um "string" S no seu valor inteiro (se possível) e o armazena em I (se houver erro de conversão, uma chamada à rotina ERRO(<n>), que siga imediatamente a referência a CHARINT, será executada);

- NOVOROT(ROT:STRING) - gera e atribui a ROT um novo rótulo numérico, como os de FORTRAN e Pascal (esta geração de rótulos começa em '1', de modo que o primeiro rótulo gerado é '1', e o último é '9999'; este mecanismo pode ser usado para se gerar variáveis na linguagem objeto, concatenando-se aos rótulos caracteres alfabéticos);

- CAT(S1:STRING, S2:STRING, S3:STRING) - atribui a S3 a concatenação de S1 e S2, nesta ordem;

- SUC(I:INTEIRO) - soma 1 ao inteiro I;

- ATR(X:inteiro-ou-string, V:tipo-de-X) - atribui o valor V à variável X.

A inserção de novas regras ou semas (rotinas semânticas) é feita através do comando INCLUI, que faz o envelope gravar na biblioteca a próxima regra (ou sema) especificada pelo usuário.

Para retirar da biblioteca uma regra ou sema, usa-se o comando EXCLUI, que tem um identificador como argumento, o qual corresponde ao nome da regra (metavariável sob definição) ou sema a ser excluído.

Ainda quanto ao uso da biblioteca, note-se que o usuário pode estabelecer para uso privado uma biblioteca alternativa. Para tanto deverá iniciar uma sessão com o comando CRIA com a opção BIBLIO e, a partir daí, com o comando INCLUI, agregar suas regras e rotinas. Ao ser citada, numa próxima sessão, esta nova biblioteca passa a ser a biblioteca de trabalho, substituindo a biblioteca STANDARD.

4.3 Diagramas de Fluxos de Dados de Nível Mais Alto

Na figura 4.2 são dados os DFD's que procuram detalhar um pouco mais a figura 4.1.

DFD nível 0 contextual

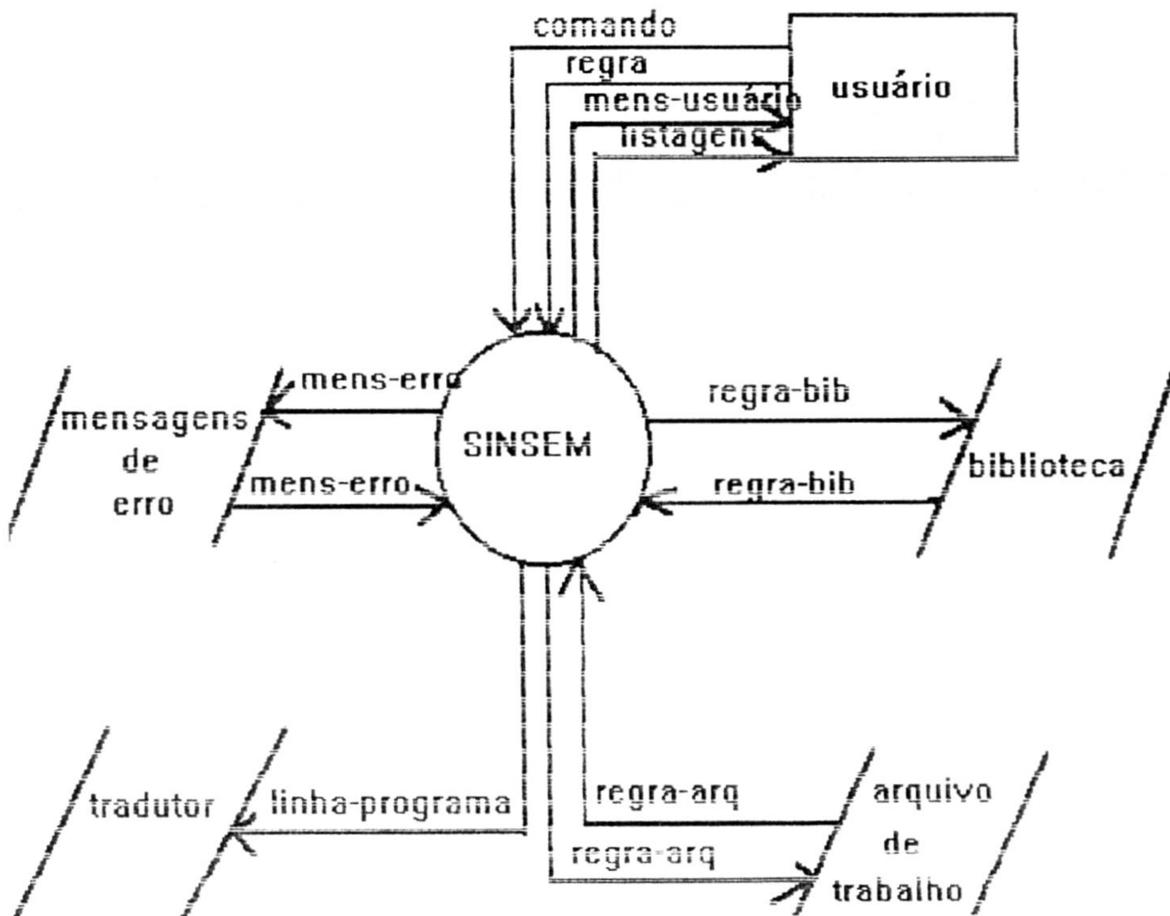


Fig. 4.2 DFD's de SINSEM

DFD nível 1 SINSEM

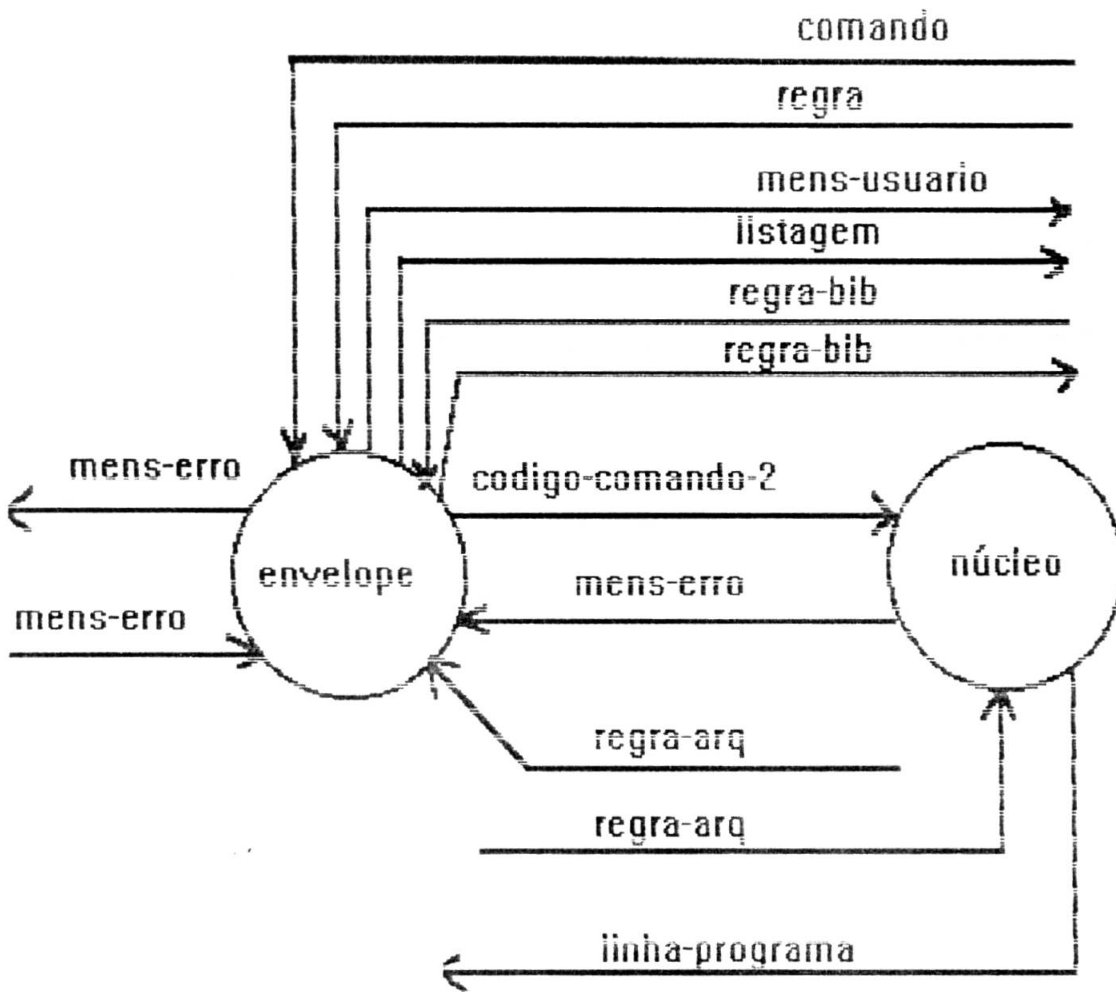
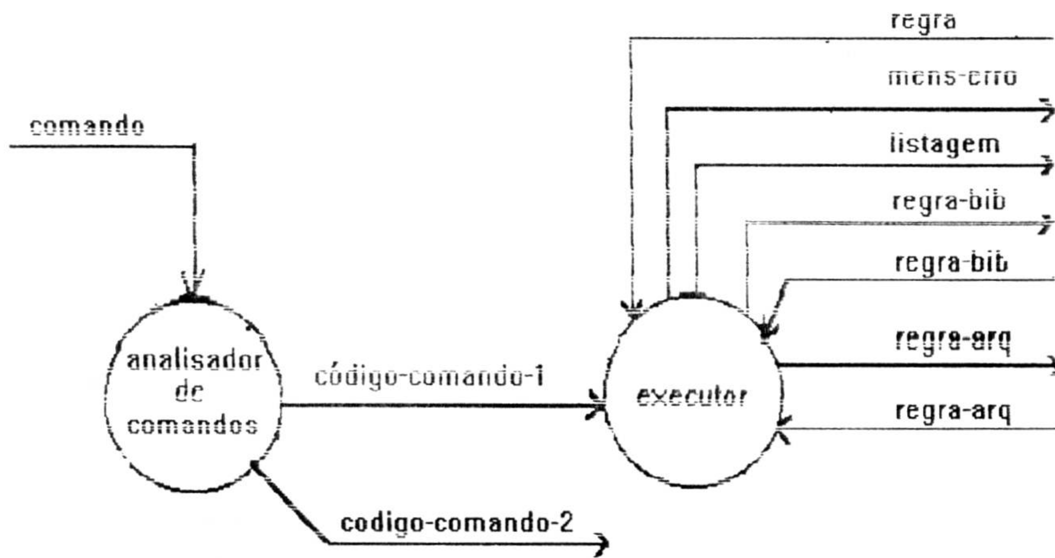


Fig. 4.2 (cont.) DFD's de SINSEM

DFD nível 2 envelope



DFD nível 2 núcleo

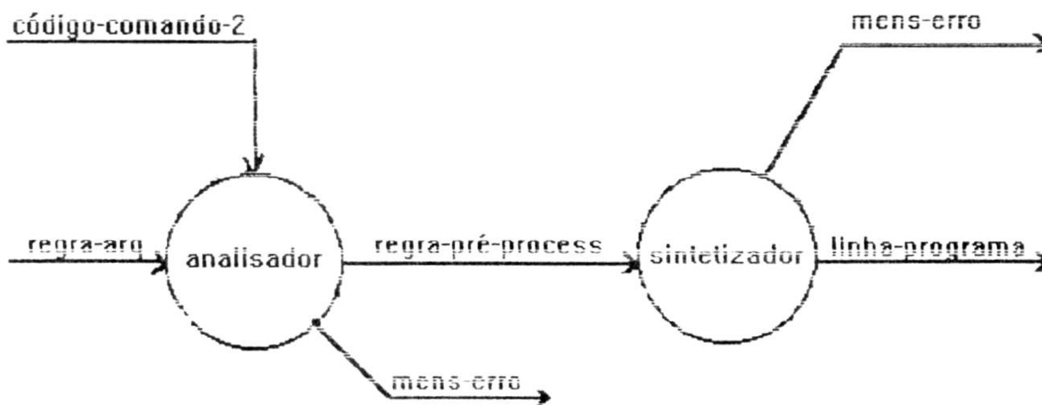


Fig. 4.2 (concl.) DFD's de SINSEM

TABELA 4.2 Dicionário de Dados Resumido

- Obs.: Todos os elementos a seguir descritos são cadeias de caracteres

comando	- um dos comandos da tabela 4.1
regra	- uma regra (ou produção), construída conforme o item 3.3.2. c
mens-usuário	- comunicação feita pelo sistema ao usuário
listagens	- arquivo impresso, contendo a interação com o usuário (comandos enviados e mensagens produzidas)
regra-bib	- uma regra a ser armazenada ou recuperada da biblioteca
regra-arq	- uma regra a ser armazenada ou recuperada da especificação que está sendo construída
linha-programa	- um trecho do programa tradutor que já possa ser armazenado
mens-erro	- mensagem de erro
código-comando-1	- código de comando a ser enviado ao processo "executor" (comando do usuário que não envolve atividades de construção
código-comando-2	- código de comando a ser enviado ao processo "analisador" (comando que envolve atividades de construção do tradutor)
regra-pré-process	- regra transformada pelo processo "analisador" de modo a ser operada pelo processo "sintetizador" (que produz linhas de programa)

5 EXEMPLO DE USO: CSSD

Neste capítulo se apresenta. a título de ilustração. a especificação de uma linguagem de programação. CSSD /MAR83/. A semântica dinâmica é dada em função de instruções de baixo nível. que correspondem ao conjunto pertencente ao interpretador subjacente ao sistema CSSD. Esta linguagem é modular. no sentido de que estabelece níveis para declarações de variáveis. Na especificação a seguir. usa-se três pilhas para compor a tabela de símbolos. a qual só guarda o nome da variável e sua localização em termos de dois índices: BN. que dá o nível de bloco. e ON. que dá o deslocamento da variável (como um número de ordem) em relação ao início do registro de ativação de cada bloco (a pilha LOCAIS). As variáveis da linguagem são todas inteiras. não havendo outro tipo. o que dispensa armazenamento de atributos.

Os comandos são os seguintes(dando-se sua forma geral e a semântica informal):

a) atribuição

<id> := <expr>

As expressões admitem operadores aritméticos(+. -. *. DIV. REM) e relacionais(=. #. <. <=. >. >=). são avaliadas da esquerda para a direita tendo prioridades usuais(relacionais por último). podendo-se usar

parênteses. Os valores 1 e zero são tomados como verdadeiro e falso, respectivamente.

b) bloco

NEW <declarações> BEGIN <comandos> END

Estabelece ambientes locais. Um programa é um bloco.

c) alternativa simples e dupla

IF <expr> THEN <comando> [ELSE <comando>] END IF

d) iteração com condição testada antes

WHILE <expr> DO <comandos> END WHILE

e) chamada de procedimento

<id> [(<lista-de-expressões>)]

A passagem de parâmetros é feita por valor. Uma curiosidade é que como argumento de um procedimento pode aparecer a própria declaração de um procedimento. O parâmetro correspondente deve ser usado como tal. (As regras 15 a 18 a seguir detalham isto.)

As declarações podem ser de variáveis ou de procedimentos. Todas as variáveis devem ser declaradas. O único tipo disponível é o inteiro. Seus formatos são os seguintes:

VAR <lista-de-identificadores> ;

PROC <id> [(<lista-de-identificadores>)] ; <bloco>

Todo bloco ocasiona a alocação de um registro de ativação, que funciona como pilha, onde são empilhados inclusive os registros de ativação de blocos mais internos. Os identificadores entram na tabela de símbolos com dois atributos somente(que não se relacionam com tipos): BN, que dá o número do bloco e ON, que dá seu deslocamento ordinal dentro do bloco.

As instruções da máquina virtual subjacente são as seguintes:

a) instruções logico-aritméticas

Não têm operando explícito. Têm o topo da pilha como operando esquerdo e seu antecessor como operando direito. O resultado é guardado no operando direito e o topo é desempilhado.

ADD	adição
SUB	subtração
MUL	multiplicação
DIVI	divisão inteira
REM	resto da divisão inteira
EQ	igual
NE	não-igual
LS	menor que

LE menor ou igual

GT maior que

GE maior ou igual

b) desvios

JMP L desvio incondicional para o endereço L

JIF L desvio se o topo é zero, com retirada do topo da pilha

JMS L desvia para o endereço L, guardando no topo da pilha o endereço da próxima instrução

c) movimento de dados

LD BN, ON "load"

ST BN, ON "store"

LDC indice "load constant"

O topo da pilha funciona como registrador para "load". As constantes são armazenadas numa área fixa da pilha,correspondendo ao bloco mais externo,fora dos registros de ativação.

d) manipulação dos registros de ativação

BLK BN, length entrada de bloco, com length dentificadores

BEX saida de bloco, liberando a área

ENT BN, n entrada de procedimento,com n argumentos

RTN retorno de procedimento

MRK prepara espaços na pilha para a chamada de procedimento

CALL chamada de procedimento.

Para a especificação que é dada a seguir, admite-se que as instruções sem operandos podem usar 1 palavra de código; as de 1 operando, 2 palavras; as de 2 operandos, 3 palavras. Uma palavra será representada por 4 caracteres de um arquivo Pascal do tipo file of char.

A seguir, a especificação de CSSD, com as 5 primeiras regras (e mais a regra 18) comentadas.

LINGUAGEM CSSD.

BIBLIOTECA STANDARD.

SINTAXE

```
VARTRAB VARS:PILHA-STR;
     BLOCOS:PILHA-INT;
     LOCAIS:PILHA-INT;
     END:INTEIRO.
```

REGRA 1:

```
 <prog> --> $ATR(END,0)
        <stmt>
```

"Um programa é um comando. Sempre é atribuído 0 como endereço relativo inicial para contagem dos endereços."

REGRA 2:

```
 VARTRAB BN:INTEIRO;
     L: INTEIRO;
     K: INTEIRO.
 <stmt> --> 'NEW' $ATR(BN,0)
        <block>
```

"Foram declaradas as variaveis BN, que vai ser o contador dos blocos (níveis); L, K para rótulos."

"Um comando pode ser um bloco começando por NEW, quando se tratará de um novo programa(por isso, BN recebe 0)."

OU 'IF'<expr> $NOVOROT(L)

$GEN2('JIF',L)

'THEN' <stmtlist>

<rest-if>

"Um comando pode ser um alternativa ('if-then-else') simples ou dupla, terminando com um delimitador especial, END IF. Depois de se avaliar a expressão (falso = 0, verdadeiro /= 0), é gerado um rótulo para o salto sobre o 'then', e é gerada a instrução ' JIF L ' (salto se o topo da pilha é zero). Para isso é usada a rotina GEN2 que dá saída a dois campos sobre o código objeto e incrementa o endereço relativo em 2. A segunda parte da alternativa está especificada na regra <rest-if>."

OU 'WHILE' $NOVOROT(K)

$NOVOROT(L)

$ATR-ROT(L)

<expr> $GEN2('JIF',K) 'DO'

<stmtlist>

'END WHILE' $GEN2('JMP',L)

$ATR-ROT(K)

"São reservados dois novos rótulos, agora para o 'while', e um deles (aquele usado para marcar o ponto de retorno do laço) já recebe o endereço relativo atual do código objeto. Depois de avaliada a expressão de controle, deve

ser testado seu resultado, o que é feito com 'JIF K', sendo K o rótulo que vai ser atualizado após ter sido produzido o código da lista de comandos interna, ou seja, depois do delimitador 'END WHILE', e ainda depois de já ter sido produzido o código para o retorno para o teste, o qual se faz via um desvio incondicional ('JMP L')."

OU <identificador> <id-option>

"Um comando pode comecar por um identificador, quando então poderá ser uma atribuição ou uma chamada de procedimento."

FIM

REGRA 3:

VARTRAB L1:INTEIRO.

<restif> --> 'END IF' $ATR-ROT(L)

"Caso nao haja ELSE, basta atualizar o rótulo criado para o salto do THEN."

OU 'ELSE' $NOVOROT(L1)

$GEN2('JMP',L1)

$ATR-ROT(L) <stmtlist>

'END IF' $ATR-ROT(L1)

"Caso haja ELSE, NOVOROT cria um novo rótulo para estabelecer o salto sobre o ELSE no caso da execução do THEN. Este salto é gerado pela instrução 'JMP L1'. O valor de L1 só será conhecido depois de se ter processado os comandos do ELSE, ou seja, após o 'END IF'. Antes de

processar a lista de comandos do ELSE, o rótulo para o salto sobre o THEN é atualizado."

FIM

REGRA 4:

VARTRAB ON:INTEIRO.

<id-option> --> ':=' $RETRIEVE(<identificador>,BN,ON)

<expr> GEN3('ST',BN,ON)

"No caso do comando de atribuição, o identificador é pesquisado na tabela de símbolos, obtendo-se seus atributos de localização: número de nível e deslocamento local(BN, ON). A instrução ST armazena o resultado da avaliação da expressão na posição correspondente da pilha dos registros de ativação. GEN3 produz o codigo 'ST BN, ON' que tem 3 locações de extensão."

OU $RETRIEVE(<identificador>,BN,ON)

$GEN3('LD',BN,ON)

$GEN1('MRK') <actualpar> $GEN1('CALL')

"Na chamada de um procedimento, ele é localizado na tabela de símbolos. Seu endereço é carregado no topo da pilha. A instrução MRK abre espaços para uso da instrução CALL, que estabelece o endereço de retorno na pilha e executa o desvio, usando o que está agora no topo."

FIM

REGRA 5:

VARTRAB ON:INTEIRO;

LENGTH:INTEIRO.

<block> --> $SUC(BN) <vardcl>

<procdcl> $GEN3('BLK',BN,LENGTH)

'BEGIN'

<stmtlist>

'END' $GEN1('BEX')

$DECR(BN)

FIM

"Um novo bloco, a primeira coisa que faz é incrementar o nível do bloco. A declaração de variáveis é o que vem a seguir (opcional), seguida ou não de declarações de procedimentos, que por sua vez contêm blocos; mas, antes de iniciar-se a lista de comandos local ao bloco, abre-se espaço na pilha para as variáveis locais, mediante a instrução BLK, que localiza o bloco pelo seu parâmetro BN, e usa LENGTH (computada em <id-list>) para reservar este espaço. Ao terminar o bloco, esta área é liberada (pela instrução BEX) e o nível de bloco é decrementado pela rotina DECR."

REGRA 6:

<vardcl> --> 'VAR' $ATR(LENGTH,0)

<id-list> ';'

OU <vazio>

FIM

REGRA 7:

<id-list> --> <identificador>

$SUC(LENGTH)

$ATR(ON,LENGTH)

$INSERT(<identificador>,BN,ON)

FIM

REGRA 8:

<id-list-tail> --> ',' <id-list>

OU <vazio>

FIM

REGRA 9:

<procdcl> --> <proc> <procdcl-tail>

OU <vazio>

FIM

REGRA 10:

<procdcl-tail> --> <procdcl>

OU <vazio>

FIM

REGRA 11:

```
VARTRAB ON:INTEIRO;
      LENGTH:INTEIRO.
<proc> --> 'PROC' <identificador>
        $INSERT(<identificador>,BN,ON)
        $NOVOROT(L)
        $GEN2('JMS',L)
        <formalpar> ';' $GEN3('ENT',BN,LENGTH)
                           $ATR-ROT(L)
        <block> ';' $GEN1('RTN')
                       $GEN3('ST',BN,ON)
FIM
```

REGRA 12:

```
<formalpar> --> '(' $ATR(LENGTH,0) <id-list> ')'
OU          <vazio>
FIM
```

REGRA 13:

```
<stmt-list> --> <stmt> <stmtlist-tail>
OU         <vazio>
FIM
```

U F R G S
INSTITUTO DE INFORMÁTICA
BIBLIOTECA

REGRA 13:

<stmtlist-tail> --> ';' <stmtlist>

OU <vazio>

FIM

REGRA 15:

<actualpar> --> '(' <expr-list> ')'

OU <vazio>

FIM

REGRA 16:

<expr-list> --> <expr-atom> <expr-list-tail>

FIM

REGRA 17:

<expr-list-tail> --> ',' <expr-list>

OU <vazio>

FIM

REGRA 18:

VARTRAB L:INTEIRO.

<expr-atom> --> 'PROC' $SUC(BN)

$NOVOROT(L)

$GEN2('JMS',L)

<formalpar> $ATR-ROT(L)

'BEGIN' $GEN3('ENT',BN,LENGTH)

<stmtlist>

'END' $GEN1('RTN')

$DECR(BN)

$POPTOTAL(LOCAIS)

OU <expr>

FIM

"Comentamos aqui o caso de um procedimento ser argumento de outro. E' a primeira opção da regra acima. Ao encontrar 'PROC' como anunciador de uma tal situação numa lista de argumentos, o sistema já incrementa o nível de bloco, para preparar a avaliação do procedimento-argumento, estabelece um rótulo para saltar a área de parâmetros e o salto. Após ter estabelecido a área de parâmetros (regra <formalpar>), abre espaço para eles, com a instrução ENT (o tamanho é calculado em <formalpar>). Antes do bloco interno se atualiza o rótulo L. Depois dele, se estabelece o retorno e o decremento do nível de bloco, liberando-se a área local."

REGRA 19:

<expr> --> <term> <expr-tail> FIM

REGRA 20:

<expr-tail> --> <relop> <expr>

OU <vazio> FIM

REGRA 21:

<term> --> <factor> <term-tail> FIM

REGRA 22:

<term-tail> --> <adop> <term>

OU <vazio> FIM

REGRA 23:

<factor> --> <primary> <factor-tail> FIM

REGRA 24:

<factor-tail> --> <mulop> <factor>

OU <vazio> FIM

REGRA 25:

VARTRAB VAL:INTEIRO;

BN:INTEIRO;

ON:INTEIRO.

<primary> --> '(' <expr> ')'

OU <inteiro> $CHARINT(<inteiro>,VAL)

$GEN2('LDC',VAL)

OU <identificador> $RETRIEVE(<identificador>,BN,ON)

$GEN3('LD',BN,ON)

FIM

REGRA 26:

<relop> --> '=' $GEN1('EQ')

OU '~=' $GEN1('NE')

OU '<' $GEN1('LS')

OU '<=' $GEN1('LE')

```
OU          '>'  $GEN1('GT')

OU          '>=' $GEN1('GE')

FIM
```

REGRA 27:

```
<adop> --> '+' $GEN1('ADD')

OU          '-'  $GEN1('SUB')

FIM
```

REGRA 28:

```
<mulop> --> '*'     $GEN1('MUL')

OU          'DIV'   $GEN1('DIVI')

OU          'REM'   $GEN1('REM')

FIM.

SEMANTICA

SEMA GEN1(X:STRING);

 "gera uma instrução sem operandos "

 COD(X);

 END := END + 1

FIM
```

```
SEMA GEN2(X:STRING,Y:INTEIRO);
 "gera uma instrução de um operando"
 VAR VAL:STRING;
 COD(X);
 INTCHAR(Y,VAL); COD(VAL);
 END := END + 2
FIM
SEMA GEN3(X:STRING,Y:INTEIRO,Z:INTEIRO);
 "gera uma instrução de 2 operandos"
 VAR VAL:STRING;
 COD(X);
 INTCHAR(Y,VAL); COD(VAL);
 INTCHAR(Z,VAL); COD(VAL);
 END := END + 3
FIM
SEMA ATR-ROT(L:INTEIRO);
 "atualiza a referência a priori de um rótulo"
 L := END
FIM
```

```
SEMA INSERT(X:STRING,Y:INTEIRO,Z:INTEIRO);
 "insere X, Y e Z nas pilhas de variáveis,blocos e            locais"
 PUSH(VARS,X);
 PUSH(BLOCOS,Y);
 PUSH(LOCAIS,Z)
FIM
SEMA RETRIEVE(X:METAVAR,BN:INTEIRO,ON:INTEIRO);
 "pesquisa nas pilhas de identificadores, blocos e            variáveis locais"
 VAR I:INTEIRO;
 PESQ(VARS,X,I);
 PESQ(BLOCOS,I,BN);
 PESQ(LOCAIS,I,ON)
FIM
SEMA DECR(X:INTEIRO);
 X := X-1
FIM.
```

6 CONCLUSÕES

A possibilidade da especificação de linguagens de programação de uma forma que integre sua definição formal e sua implementação ficou clara com este trabalho. Foi proposta uma linguagem de especificação que procura relacionar o ato de programar um tradutor com o ato mesmo de especificar a linguagem, fundindo estas atividades.

O trabalho começa com a apresentação da motivação que o impulsionou. Passa para uma visão geral de um sistema de programação que seria capaz de ir ao seu encontro, buscando também estabelecer uma certa filosofia de uso do mesmo, comunicando ao possível usuário o espírito da proposta. Isto prepara a apresentação do trecho mais importante, a descrição da metalinguagem associada.

De início, procurou-se situar o sistema a ser produzido numa paisagem mais abrangente, a das metalinguagens em geral, em relação a seus objetivos e soluções encontradas, verificando-se a amplitude destas últimas. Em seguida, se fundamenta a escolha da proposta construtiva interpretativa por ser a mais próxima dessa necessidade que se coloca: relacionar definição e implementação. E, então, se descreve a metalinguagem.

Em continuação, se apresenta os componentes do sistema de programação como um todo, enfatizando sua personalidade interativa, o que se

torna cada vez mais imperioso em qualquer sistema de computação: que ele nao apenas seja usado, mas ajude a ser usado.

Por fim, se apresenta um exemplo de especificação onde se referencia algumas das características oferecidas pela metalinguagem, não se tendo no entanto explorado todas as suas potencialidades.

Daqui em diante, evidencia-se que o uso de um tal sistema é que pode consolidar ou desaprovar as opções feitas, não só quanto ao método adotado, mas, principalmente, quanto ao ferramental que o compõe. Especificamente, citem-se as rotinas sintáticas e semânticas pré-definidas, que merecem um maior estudo quanto `a determinação de um conjunto mais significativo.

Outra tarefa que se impõe é o desenvolvimento de um trabalho de implementação, que se encontra apenas esboçado aqui. Deste trabalho de implementação se poderá partir para reformulações muito importantes em direção ao incremento da proposta, com vistas a torná-la verdadeiramente operacional, o que justificaria o esforço empreendido até agora.

BIBLIOGRAFIA

[AHO72] AHO, Alfred ; ULLMAN, Jeffrey. **The theory of parsing, translation and compiling.** Englewood Cliffs: Prentice Hall, 1972. 2v. 1002 p.

[DON72] DONOVAN, John J. **Systems programming.** Tokyo: McGraw-Hill Kogakusha, 1972. 488 p.

[FEL68] FELDMAN, Jerome ; GRIES, David. Translator writing systems. **Communications of the ACM.** New York: v. 11, n. 2, p. 77-113, Feb. 1968.

[GHE85] GHEZZI, Carlo ; JAZAYERI, Mehdi. **Conceitos de linguagens de programação.** Rio de Janeiro: Campus, 1985. 306 p.

[HAM84] HAMMOND, K. ; RAYWARD-SMITH V. J. A survey on syntactic error recovery and repair. **Computer Languages.** Exeter, v. 9, n. 1, p. 51-67, Jan. 1984.

[HOA74] HOARE, C. A. R. ; LAUER, P. E. Consistent and complementary formal theories of the semantics of programming languages. **Acta Informatica.** Berlin, v. 3, n. 2, p. 135-153, Apr. 1974.

[HOL82] HOLT, R. C. et al. An introduction to S/SL:Syntax/Semantic Language. **ACM Transactions on Programming Languages and Systems**. Baltimore, v.4, n. 2, p. 149-178, Apr. 1982.

[JOH78] JOHNSON, Stephen C. ; LESK, M.E. Language development tools. **The Bell System Technical Journal**. Murray Hill, v. 57, n. 6, p. 2155-2175, July-Aug. 1978.

[KAS82] KASTENS, Uwe et al. **GAG: a practical compiler generator.** Berlin: Springer-Verlag, 1982. (Lecture Notes in Computer Science, 141).

[LEC82] LECARME, Olivier et al. Computer-aided production of language implementation systems: a review and a classification. **Software-Practice and Experience**. London, v. 12, n. 9, p. 785-824, Sept. 1982.

[MAR76] MARCOTTY, Michael; LEDGARD, Henry F. ; BOCHMANN, Gregor V. A sampler of formal definitions. **Computing Surveys**. New York, v. 8, n. 2, p. 191-276, June 1976.

[MAR83] MARTINS, Ademir da Rosa. **Linguagem CSSD**. Porto Alegre, CPGCC-UFRGS, 1983.

[MEE81] MEERTENS, Lambert. Issues in the design of a begginer's programming language. In: BAKKER, J. W. ; VAN VLIET, J.C., eds. **Algorithmic languages**. Amsterdam, North Holland, 1981. pag. 167-184.

[TEN76] TENNENT, R.D. The denotational semantics of programming languages. **Communications of the ACM**, New York, v. 19, n. 8, p. 437-453, Aug. 1976.

[WEG72] WEGNER, Peter. The Vienna Definition Language. **Computing Surveys**, New York, v. 4, n. 1, p. 5-63, Mar. 1972.

[WIL77] WILLIAMS, John H. ; FISHER, David A., eds. **Design and implementation of programming languages**. Berlin: Springer-Verlag, 1977. (Lecture Notes in Computer Science, 54).

[WIL78] WILLIAMS, M.H. ; BULMER, A.R. Use of a formal notation for static semantics in compiler design. **Software Practice and Experience**. London, v. 8, n. 5, p. 579-584, May 1978.

[WIL80] WILLIAMS, M. Howard. A formal notation for specifying static semantic rules. **Computer Languages**. Exeter, v. 5, n.1, p. 37-55, 1980.

[WIL81] WILLIAMS, M. Howard. Methods for specifying static semantics. Computer Languages. Exeter. v. 6. n. 1. p. 1-11. June 1981.

[WIL82] WILLIAMS, M. Howard. A flexible notation for syntactic definitions. ACM Transactions on Programming Languages and Systems. Baltimore. v.4. n. 1. p. 113-119. Jan. 1982.

[WUL80] WULF. William A. PQCC: a machine relative compiler technology. In: COMPSAC80. Chicago. Oct. 27-31. 1980. Proceedings. Los Alamitos. IEEE Computer Science Press. 1980. p. 24-36.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ESPECIFICAÇÃO DE UM SISTEMA DE SUPORTE
À IMPLEMENTAÇÃO DE LINGUAGENS DE PROGRAMAÇÃO

Dissertação apresentada aos Srs.

____

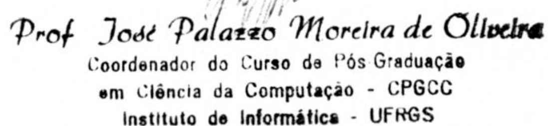
____

____

____

Visto e permitida a impressão

Porto Alegre, 29/12/94



____

Prof. Roberto Tom Price
Coordenador do Curso de
Pós-Graduação em Ciência da Computação