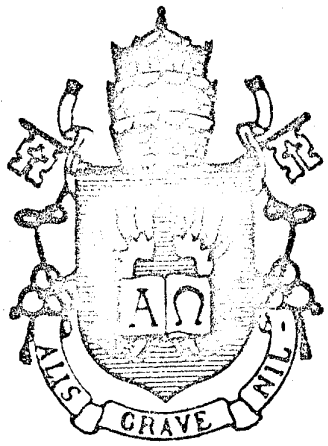


PUC

Série: Monografias em Ciência da Computação,
No. 6/91

AS TABELAS DE SÍMBOLOS E AS LINGUAGENS ORIENTADAS A OBJETOS

Werther J. Vervloet

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

PUC RIO - DEPARTAMENTO DE INFORMÁTICA

Série: Monografias em Ciência da Computação, No. 6/91
Editor: Carlos J. P. Lucena Maio, 1991

AS TABELAS DE SÍMBOLOS E AS LINGUAGENS ORIENTADAS A OBJETOS *

Werther J. Vervloet

* Trabalho patrocinado pela Secretaria de Ciência e Tecnologia da Presidência da República.

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC Rio - Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453 - Rio de Janeiro, RJ
Brasil

Tel.:(021)529-9386 Telex:31078 Fax:(021)511-5645
E-mail:rosane@inf.puc-rio.br

Abstract:

Symbol tables (ST's) play an important role in the construction of compilers, interpreters and other systems that parse source code. In this report the requirements for ST's in object oriented environments are discussed, and ST implementations for the programming language TOOL is presented. The use of aggregate structures to simplify the implementation of some ST functions is also shown. Performance considerations are presented.

Keywords:

Symbol tables, object oriented languages.

Resumo:

As tabelas de símbolos são parte vital na construção de compiladores, interpretadores e ferramentas que analisam software fonte. Este trabalho discute os requisitos de uma TS num ambiente orientado a objetos e mostra uma implementação para a linguagem TOOL. Discute também como alguns mecanismos são habilitados via estruturas agregadas à TS, bem como algumas considerações sobre desempenho.

Palavras-chave:

Tabelas de símbolos, linguagens orientadas a objetos.

Agradecimentos

Gostaria de agradecer a todas as pessoas que direta ou indiretamente contribuiram na confecção deste trabalho. Gostaria também de agradecer especialmente aos professores Sergio Carvalho e Otavio Pecego Coelho, à SPA Sistemas Planejamento e Análise SA e à Universidade do Estado do Rio de Janeiro, UERJ, pela colaboração recebida, sem a qual, este trabalho não seria possível.

SUMÁRIO

1.0 - Introdução.

Tabelas de Símbolos (TS's) são complexas estruturas de informação, usadas por analisadores de programa fonte, para armazenar e recuperar nomes (símbolos) e seus respectivos atributos, introduzidos por programadores ou definidos na linguagem fonte.

A complexidade de uma TS para um analisador A de uma linguagem L é consequência tanto da complexidade de L quanto dos objetivos de construção de A.

Dentre os fatores que afetam a complexidade de uma linguagem podem ser citados:

- regras de visibilidade para nomes;
- regras para a amarração de tipos;
- separação léxica entre nomes e atributos.

Na construção de um analisador de programas de nível profissional, mais aspectos devem ser considerados no projeto de uma TS:

- o espaço utilizado pela TS;
- o tempo de resposta para inserção e/ou consulta de nome/atributo;
- a capacidade de crescimento da TS;
- os serviços que devem ser prestados ao analisador (e, no caso de compiladores, ao gerador de código).

Este trabalho descreve uma implementação de TS para uma linguagem de programação (LP) orientada a objetos (LOO). Seu objetivo principal é mostrar as dificuldades adicionais que surgem no projeto e implementação da TS, causadas pelas características básicas de LOO's. Algumas soluções adotadas no projeto e implementação de um compilador, ora em andamento, são discutidas em detalhe.

Na Seção 2 apresentamos princípios básicos em orientação a objetos de forma resumida, de forma a se criar fundamentos para o discurso que se segue. As implicações destes princípios no projeto de uma TS são aí apresentadas.

Na Seção 3 apresentamos a implementação de uma TS para o compilador da linguagem TOOL. À guisa de introdução da seção, descrevemos sumariamente o ambiente de execução e a LOO fonte.

2.0 - Orientação a objetos.

Temos como noção básica de orientação a objetos a existência de CLASSES como modeladores dos objetos. Cada classe define uma estrutura de dados que chamamos de Representação (REP) para seus objetos e uma coleção de procedimentos (Métodos) aplicáveis aos objetos desta classe. A cada campo do REP bem como a cada método são atribuídos nomes pelos quais são identificados. Estes nomes podem ou não ser exportados para uso em outras classes. Nomes cujo acesso é permitido de fora de seu ambiente, são chamados de PÚBLICOS ou VISIVEIS, à semelhança da declaração **PUBLIC** em C++ [STROUSTRUP 86].

Recentemente, LOO's voltadas para compilação nos sugerem a utilização de verificação estática de tipos [AMERICA 89], [BUHR 88], [SCHAFFERT 86] e [MEYER 88] com vistas à geração de código robusto e eficiência na execução, evitando sempre que possível o adiamento da detecção de erros para a fase de execução. Para estas, alguns aspectos inerentes à orientação para objetos afetam diretamente o projeto das TS's:

- Herança.
- Composição, Comunicação e Localização de Objetos.
- Polimorfismo.
- Definição adiada de métodos (métodos "deferred" ou virtuais).
- Aninhamento de classes.

Outro ponto a considerar é a necessidade de se pensar em bibliotecas de programas, tanto a nivel de classes já compiladas (código objeto), como a nível de nomes exportados pelas classes e seus atributos. Uma biblioteca de classes já compiladas habilita confortavelmente a compilação em separado de novas classes e sua posterior "Link Edição". Uma biblioteca contendo as especificações exportadas pelas classes é de vital importância para a verificação estática das chamadas a métodos, sem o que, a tarefa de se programar fica bastante tediosa, uma vez que está condicionada à presença do código fonte de todas as classes referenciadas.

A seguir, uma rápida introdução a cada um destes aspectos é acompanhada de uma análise sobre seu reflexo na construção da TS. Nos exemplos usamos sintaxe bastante próxima à de TOOL.

2.1 - Herança.

Consideramos *Herança* um aspecto básico de orientação a objetos já que pode ser encontrado na maioria das LOO's existentes.

Este aspecto consiste em permitir que ao se definir uma nova classe, possamos importar para esta, as definições dos REP's de outras classes, bem como o acesso aos métodos destas. Quando esta "importação" é feita de uma única classe, chamamos herança simples, quando feita de mais de uma classe, chamamos herança múltipla.

Discutiremos aqui principalmente a herança simples uma vez que TOOL não faz herança múltipla. Observações sobre este e outros aspectos não constantes da linguagem TOOL podem ser encontradas no final desta seção.

Quando uma (sub) classe C é construída a partir de outra (super) classe D, diz-se que C herda a representação dos objetos (REP), os métodos e as variáveis de classe de D. Isto significa que ao compilar uma classe construída desta forma (classe derivada), o compilador deve ter em mãos as informações sobre a classe da qual é feita a herança.

☐ *Definição 1.*

Uma classe C derivada de D implica em D ser ***superclasse*** de C. DERIV (C,D) implica SUPER (D,C).

☐ *Definição 2.*

Uma classe C descende de D, DESCENDENTE (C,D), quando SUPER (D,C) ou se existe uma classe C1 tal que SUPER (C1,C) e DESCENDENTE (C1,D).

Exemplo 1:

```
CLASS C FROM D
  REP       -- Representação de memória para objetos de C
            -- além da já herdada de D
    INT i;
    ARRAY
      x OF [10] REAL;
    END ARRAY
  END REP
```

```
  ...
END CLASS
CLASS D
  REP          -- Representação de memória para objetos de D
               -- (neste caso não temos herança)
    STRING nome;
    BOOL i;
  END REP
  ...
END CLASS
```

Neste exemplo temos a relação SUPER (D, C).

Uma definição do tipo:

```
C obj1;
```

define um objeto da classe C (obj1) cujo REP contém (por herança) o objeto "i" da classe dos inteiros e o array "x [10]" de objetos da classe dos reais. Contém também (agora por definição) o objeto "nome" da classe dos Strings e o objeto "i" da classe dos Booleans.

2.1.1 - Reutilização de nomes.

Algumas LOO's permitem que subclasses "vejam" os nomes dos componentes herdados de suas superclasses. TOOL permite também a reutilização destes, isto é, criar no REP de uma subclasse um objeto com o mesmo nome de outro definido no REP da superclasse. Podemos observar este aspecto no exemplo acima.

2.1.2 - Redefinição de métodos.

Quando uma classe C, define um método M e já existe ao menos um método M definido em uma de suas classes ascendentes, temos uma redefinição de M em C.

Se acrescentarmos à definição da classe D no exemplo anterior:

```
METHOD mc2 (IN INT k) RETURNS BOOL;
-- método da classe D com 1 parâmetro e retorno.
  ...
END METHOD
METHOD md1;
```

```
-- método da classe D sem parâmetros nem retorno.
  ...
END METHOD
```

E também à classe C:

```
METHOD mc1 (IN INT j, INOUT REAL y) RETURNS REAL;
-- método da classe C com 2 parâmetros e retorno.
...
END METHOD
METHOD mc2 (IN INT j) RETURNS BOOL;
-- método da classe C (redefine mc2 herdado de D)
-- com 1 parâmetro e retorno.
  ...
END METHOD
```

Ao objeto "obj1" definido acima (classe C) são aplicáveis os métodos md1 (herdado), mc1 (definido) e mc2, este redefinido em C. Embora não seja padrão nas LOO's, em TOOL, métodos redefinidos devem conservar a mesma ASSINATURA (nome do método, classe e modo dos parâmetros e classe do objeto retornado). As razões desta restrição estão comentadas adiante em 2.3.3 (Polimorfismo).

2.1.3 - Qualificação de nomes.

Ao se referenciar um nome, seja ele objeto ou método, na existência de mais de um, prevalece o nome local à classe que está sendo compilada. A maneira de se acessar os demais é prefixá-los com o nome da classe a que pertencem.

Assim,

```
C <- D.mc2 (3)
```

está se referindo ao método mc2 definido em D enquanto

```
C <- mc2 (3)
```

chama mc2 definido em C. Da mesma forma, se estamos compilando um método M definido em C, o comando

```
D.i := 0;
```

está se referindo ao i definido no REP de D e herdado em C, enquanto

```
i:= TRUE;
```

se refere ao i local ao REP de C.

A herança traz como consequências imediatas ao projeto da TS a necessidade de se implementar uma estrutura capaz de, para toda classe, recuperar as informações herdadas de sua superclasse, da superclasse desta e assim sucessivamente até a classe que originou esta "família", em outras palavras, até encontrar uma superclasse que não tenha herança.

Esta estrutura deve também tratar adequadamente a coexistência de nomes iguais em contextos diferentes, de forma a recuperar corretamente o método ou o objeto referenciado.

2.2 - Composição, Comunicação e Localização de objetos.

Um objeto pode estar definido:

- Como componente da área comum (SHARED) aos objetos da classe.
- Como componente do REP de uma classe.
- Como variável local a um método.
- Como parâmetro formal de um método
- Como retorno de um método.

Quando uma classe C define em sua representação, objetos de outras classes C1 .. Cn, diz-se que C é uma composição de C1 .. Cn.

☐ *Exemplo 2:*

```
CLASS C
  REP       -- definição por composição
    C1 Obj1;
    C2 Obj2;
     ...
    Cn Objn;
  END REP
  ...
END CLASS
```

Neste caso, o REP de C é uma sequência dos REP's de C1, C2, ... , Cn, e todos os métodos visíveis de Ci são aplicáveis a Obji (1 <= i <= n).

☐ *Exemplo 3:*

```
CLASS C
   ...
  METHOD m (.....);
     C1 obj1;
       ...
  END METHOD
  METHOD p (INOUT C1 obj) RETURNS INT i
  -- definição como parâmetro (comunicação)
    ...
  END METHOD
    ...
END CLASS
```

Também neste caso devemos notar que tanto a obj1 (local a m) quanto a obj (parâmetro formal de p) são aplicáveis os métodos visíveis de C1.

Como consequências imediatas à construção da TS, temos a necessidade de um mecanismo que controle a visibilidade no acesso às informações das classes dos objetos referenciados. Quando um objeto de uma classe A é definido numa classe B em quaisquer das situações citadas, não havendo a relação DESCENDENTE (B, A), apenas o que que está definido como visível (público) em A pode ser referenciado em B. Este conceito é aplicável tanto aos métodos quanto aos campos do REP.

2.3 - Definição, declaração e criação de objetos.

Ao se declarar um objeto, dois aspectos devem ser considerados: sua vida e seu escopo de visibilidade.

2.3.1 - Vida dos objetos.

Quanto à vida, LOO's apresentam quatro possibilidades:

- **Persistentes.**

Objetos persistentes tem sua vida independente da execução dos programas onde são usados. Normalmente só são destruídos por comandos explícitos das linguagens ou do ambiente.

- **Estáticos.**

Os objetos existem durante toda a execução do programa. Em LOO's compiladas, estes objetos são normalmente declarados em um bloco "Global", ou no bloco compartilhado pelos objetos de uma classe (SHARED OBJECTS). Pode-se também considerar como estáticos objetos através dos quais se inicia a execução de um programa.

- **Dependentes.**

Os objetos existem condicionados à vida do objeto que o contém. Objetos desta natureza são componentes de REP's de outros objetos e perduram enquanto o objeto que os declara estiver "vivo".

- **Semi-estáticos.**

Os objetos existem condicionados à execução do método que os declara. Semelhantes aos citados acima, estes objetos perduram durante a execução do método onde foram declarados. Têm a vida de uma variável local automática do ALGOL 60.

Quanto à localização, objetos persistentes, pela sua própria independência dos programas, devem residir em dispositivos de E/S, normalmente discos magnéticos. Excluindo-se estes, os demais podem ter sua alocação decidida estaticamente durante a compilação, ou dinamicamente durante a execução, no heap. Objetos estáticos (ou suas referências) devem estar alocados em

segmentos globais. Objetos semi-estáticos (ou suas referências) devem estar alocados na Pilha de Ativação. Quanto aos demais, sua localização (ou a de suas referências) deve acompanhar o objeto que os declara. Este modelo pressupõe a existência de um mecanismo de desalocação automática que garanta a destruição dos objetos quando não forem mais referenciados. LOO's normalmente possuem maneiras de se destruir explicitamente objetos no heap como em TOOL:

```
Obj <- DESTROY ();
```

2.3.2 - Visibilidade dos objetos.

Quanto ao escopo de visibilidade, pode-se adotar um modelo mais fechado, como o previsto em POOL2 [AMERICA 89], onde os acessos a componentes e métodos são exclusivos dos objetos, ou mais flexível, como em C++ [STROUSTRUP 86] e TRELLIS_OWL [SCHAFFERT 86], onde nomes podem ser visíveis a usuários de uma classe.

No presente caso, classes são sempre visíveis, bem como objetos componentes de REP's que tenham o atributo VISIBLE. Métodos são visíveis por definição a menos que tenham o atributo PRIVATE.

Nos casos de visibilidade restrita, componentes de REP's e métodos só são vistos pelos métodos de sua classe. Objetos locais a métodos só são visíveis dentro do próprio método.

2.3.3 - Polimorfismo.

Este aspecto se apresenta sempre que é permitido a um mesmo identificador se referir a objetos de classes distintas, ao longo de um programa. Normalmente este conceito é atribuído às classes. Neste trabalho, consideraremos polimorfismo como atributo dos objetos, declarado explicitamente, pois desejamos a coexistência destes com objetos não polimórficos.

Assim sendo, um objeto de uma classe é dito polimórfico quando pode assumir os comportamentos de objetos pertencentes a outras classes. Isto

significa que o REP de um objeto polimórfico, bem como os métodos aplicáveis a ele, dependem da classe atual deste objeto.

Não havendo nenhuma outra restrição, uma LOO compilada tem que forçosamente adiar a decisão de verificar a aplicabilidade de um método M a um objeto O, uma vez que em princípio a chamada,

```
O <- M (....);
```

é possível.

Uma forma de se garantir que esta é sempre válida é se restringir a abrangência do conceito de polimorfismo citado, permitindo apenas que objetos com este atributo possam assumir o comportamento de sua classe e de cada uma de suas descendentes. Para esta definição, uma estrutura adequada de TS consegue ter as informações necessárias para se promover a verificação da existência de M aplicável a O.

Ainda assim, admitindo-se a existência de parâmetros e valor de retorno para os métodos, podemos não ter a garantia durante a compilação, de que estes estão sempre corretos, pois métodos de mesmo nome podem não ter a mesma assinatura. Uma maneira de se evitar este incômodo é se obrigar, dentro do universo de métodos de mesmo nome M, aplicáveis a um mesmo objeto polimórfico O, que as assinaturas dos diversos M's sejam iguais.

Para algumas LOO's todo objeto é polimórfico como em EIFFEL [MEYER 88]. Podemos pensar também na coexistência destes com objetos que não tenham este atributo.

As justificativas para a existência destes últimos são as seguintes:

- A verificação se M é aplicável a O fica bastante simplificada.
- O gerador de código já sabe que método M deve ser chamado, o que redunda em código mais eficiente.
- Pode-se constatar que em muitas aplicações há um grande número de objetos para os quais o polimorfismo é irrelevante ou até mesmo indesejável.

☐ *Exemplo 4:*

Seja uma classe C definida com REP R e métodos M1 a Mn.

Sejam as classes C1 .. Cn definidas com REP's REP1 .. REPn respectivamente e para cada Ck (1 <= k <= n) métodos (redefinidos) M1 .. Mn, onde a relação DESCENDENTE (Ck, C) é verdadeira para todo k (1 <= k <= n).

Sejam as definições:

```
POLY C Obj;
C1 Obj1;
...
Cn Objn;
```

A instanciação

```
Obj <- NEW ();
```

cria Obj com REP da classe C.

A chamada

```
Obj <- M1 (...);
```

aplica a Obj o método M1 definido em C.

Se fizermos a seguir:

```
Obj <- ISNOW (Obj1);
```

dizemos que Obj agora "assume o papel de Obj1"

ao chamarmos

```
Obj <- M1 (...);
```

estaremos aplicando a Obj (que agora é uma instância de C1) o método M1 definido em C1.

□ *fim do exemplo 4.*

Este esquema de polimorfismo, traz para o projeto da TS a necessidade de se ter acesso às informações das classes descendentes pois a decisão de qual método deve ser aplicado a um objeto deixa de ser resolvida durante a compilação, e o gerador de código deve ter acesso a todas as alternativas possíveis (redefinições de M1) ou, em último caso, repassar a decisão para a etapa de "link edição".

2.4 - Definição adiada de métodos (Métodos "deferred" ou virtuais).

Ao se definir uma classe C, pode-se ter em mente apenas um modelador parcial para os objetos com os quais se deseja trabalhar. Em outras palavras, pode-se definir uma classe num nível de abstração onde apenas o comportamento geral dos objetos é delineado, sendo o comportamento específico destes objetos definido por suas classes descendentes.

Uma possível implementação deste conceito pode ser feita através de ***métodos de definição adiada*** (deferred). Uma classe que possui métodos deste tipo obriga a que todas suas descendentes os definam ou, mais uma vez, os repassem para suas descendentes.

Como consequência imediata, temos a necessidade de ter na TS estrutura de dados capaz de verificar se os métodos adiados numa classe são definidos ou novamente adiados por suas decendentes.

Para que um método M seja aplicável, é necessário que na árvore de aridade n que representa uma família de classes descendendes de uma mesma classe C (raiz), qualquer percurso até as "folhas" (nós sem descendência) encontre ao menos uma definição de M.

Em TOOL, classes que adiam métodos, servem apenas para modelar suas descendentes. **Nenhuma aplicação de método é válida para seus objetos.**

□ *Exemplo 5.*

```
CLASS C
  ...
  METHOD M (IN INT PARM) DEFERRED;  -- método de definição adiada
  ...
END CLASS
CLASS D FROM C
  ...
  METHOD M (IN INT P)
    ...              -- Definição do corpo do método
  END METHOD
  ...
END CLASS
```

2.5 - Observações finais sobre orientação a objetos.

2.5.1 - Herança Múltipla.

Quando uma classe **C** deriva das classes **B1 ... Bn** (n > 1), o REP de **C** contém os REP's de suas superclasses. Todos os métodos definidos em {**B1 ... Bn**} são aplicáveis aos objetos de **C**. A isto chamamos herança múltipla.

Tanto na herança simples como na múltipla, a aplicação de um método definido em uma superclasse S (SUPER (S,C)) a um objeto de **C** significa acessar apenas a parte do REP do objeto comum às duas classes. A diferença reside na localização desta parte comum, que na herança simples é sempre o início do REP do objeto.

A adoção deste mecanismo obriga à TS ter uma estrutura de DAG (Directed Acyclic Graph) para o acesso às superclasses. Na herança simples esta estrutura pode ser uma lista encadeada.

O algoritmo V para verificação de consistência da TS, no caso de herança simples, verifica apenas se as listas ascendentes são circulares. Isto nos leva ao pior caso

$$T(V) = O(2n - 1) \Rightarrow O(n)$$

onde n é o número de classes no programa.

No caso de herança múltipla, num algoritmo V' que verifique se um grafo dirigido é um DAG (i.e. não tem ciclos) tem no seu pior caso

$$T(V') = O(n!).$$ (* ver apêndice 1)

2.5.2 - Polimorfismo irrestrito.

Os conceitos de Covariância e Contravariância ([CARDELLI 88] e [AMERICA 89]) para a verificação dos parâmetros e retorno dos métodos, em caso de polimorfismo, exigem da TS estruturas ascendendes (para as superclasses) e descendentes (para as classes derivadas).

O poliformismo irrestrito como citado em 2.3.3 pode nos levar a chamadas do tipo

O <- M (...)

cuja corretude pode não ser decidível estaticamente e isto nos levaria de volta ao "late checking" que queremos evitar.

2.5.3 - Aninhamento de Classes

Alguns autores sugerem o uso de ***Classes Aninhadas*** [BUHR 88] à semelhança dos blocos aninhados de Algol 60.

Se deixarmos de lado o aspecto visibilidade, classes aninhadas e subclasses se assemelham, pois ambas tem acesso à representação das classes que as englobam (**aninhamento**) ou da superclasse (**herança**). Por outro lado, diferem fundamentalmente porque uma subclasse contém **obrigatoriamente** o REP de sua superclasse, o mesmo não acontecendo com a classe aninhada.

Com relação à composição, uma classe A é componente de outra classe B se no REP de B existe ao menos um objeto de A. Uma classe C, que contém uma outra classe D, não tem necessariamente que ter um objeto de D em seu REP.

As restrições de visibilidade inerentes ao aninhamento trazem para o projeto da TS a necessidade de mais um mecanismo de proteção, que implica na criação de novos ambientes locais.

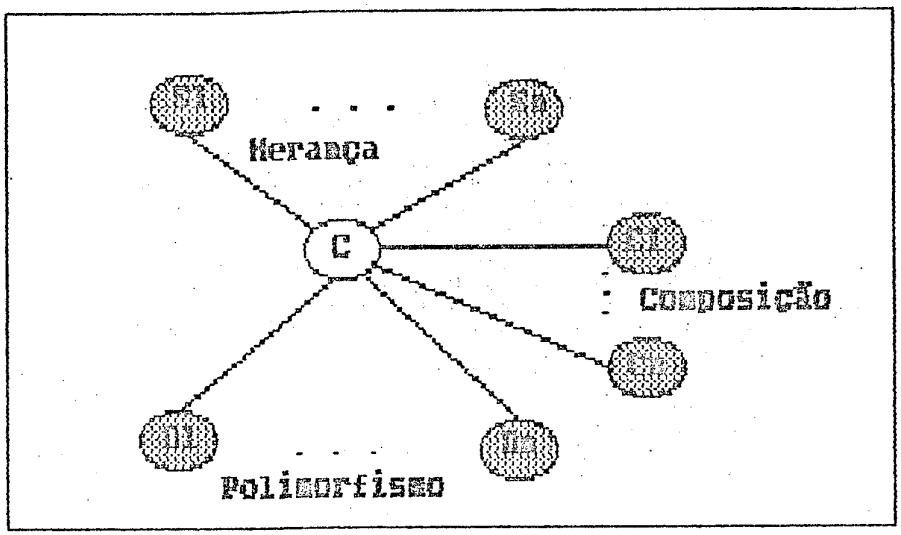
Entretanto, o uso deste tipo de construção fica extremamente restrito se observarmos as regras previstas em Algol. BUHR em seu artigo [BUHR 88] admite que estas regras são demasiadamente restritivas para um uso eficaz deste mecanismo e propõe uma abertura, tornando as classes aninhadas visíveis fora do ambiente que as define, bastando para isto que se prefixe o nome delas com o nome do ambiente que as engloba.

Neste caso, os mecanismos de herança e composição, aliados à visibilidade de componentes (como em **TOOL**) substituem o aninhamento com a vantagem de não ter que pagar o preço da criação de múltiplos ambientes locais.

> ☐ *Com respeito a este aspecto, TOOL possui apenas um mecanismo, bastante restrito em relação ao que foi exposto, que pode ser visto em 3.2.1 (IMAGE).*

2.5.4 - Complexidade da estrutura da TS.

A figura a seguir nos mostra uma estrutura possível para um nó da TS representando uma classe, no caso de herança múltipla.



Se retirarmos as restrições ao polimorfismo, a estrutura deverá ser capaz de, partindo de cada nó (classe), acessar "tudo" sobre as demais classes.

Como podemos observar, a complexidade da estrutura de dados da TS está diretamente ligada à complexidade dos mecanismos que desejamos implementar. Mesmo se tratando de um compilador, é interessante, sempre que possível, evitar mecanismos que resultem em algoritmos da família dos ***não polinomiais (NP)***, sob pena de se pagar o preço de uma compilação excessivamente demorada.

3.0 - Implementação de uma TS para uma linguagem orientada a objetos.

3.1 - O ambiente.

O projeto em questão visa implementar uma linguagem orientada para objetos voltada para equipamentos da família do IBM AT e PS2 (286 e 386) sob Microsoft Windows 3.0 e OS/2 com Presentation Manager.

A idéia principal é utilizar o próprio sistema de mensagens do Windows e PM OS/2 com duas finalidades principais:

- Promover sem esforço adicional a troca de mensagens entre os objetos de uma aplicação.
- Integrar software de diversos fabricantes, utilizando-se para tal as facilidades de uma LOO.

Nosso ambiente constará de:

- Máquina Virtual
- Biblioteca de rotinas para execução (Run Time Library)
- Administrador da Biblioteca de Classes (Source Library Manager)
- Administrador da Biblioteca de Classes compiladas (Link Library Manager)
- Link editor (Linker)
- Compilador

3.2 - A linguagem.

Uma definição formal da linguagem em questão foge ao escopo deste trabalho. Mostramos portanto apenas os aspectos relevantes ao projeto da TS. Para maiores informações sobre a liguagem TOOL sugerimos ver artigos referenciados na bibliografia.

3.2.1 - Classes.

A linguagem TOOL, para a qual projetamos a TS, tem como sua unidade de compilação a classe. Um programa nesta linguagem é uma coleção de classes. A informação de qual classe inicia o processo (o que em outras linguagens chamamos programa principal) é passada para o Linker que encarta no código a instanciação de um objeto desta como primeira instrução a ser executada.

Há 3 tipos de classes, as pré_definidas, as normais (CLASS) que seguem a orientação da maioria das LOO's e as classes extendidas (XCLASS), estas envolvendo conceitos de paralelismo e comunicação assíncrona.

As classes pré_definidas são as classes básicas encontradas na maioria das linguagens tais como inteiros (INT), reais (REAL), lógicos (BOOL), strings (STRING) etc.

Uma CLASS possui:

- Uma definição de REP que pode ser pública no seu todo ou em parte. Todo objeto de uma dada classe C tem como representação o REP definido em C, incluindo as heranças que porventura existam.
- Uma definição de área compartilhada (SHARED), visível apenas pelos métodos desta classe. Esta área tem apenas uma instância, qualquer que seja o número de objetos definidos desta classe.
- Um conjunto de métodos aplicáveis aos seus objetos, com ou sem parâmetros e retorno. Estes métodos podem ser públicos ou restritos (PRIVATE) sendo estes últimos apenas visíveis pelos demais métodos desta classe.

Uma XCLASS possui, além do REP, SHARED e métodos já descritos acima:

- Uma definição de IMAGE, que exporta um tipo.

  ☐ *Toda Xclass em TOOL permite que se associe a ela uma outra classe. Isto é feito de uma maneira bastante restrita através do IMAGE que é, na verdade, a definição do REP desta outra classe. Só é permitida a definição de um único IMAGE numa xclasse. A objetos desta natureza, aplicam-se apenas os métodos nativos de TOOL (NEW, DESTROY, SAME etc). Este mecanismo foi concebido, tendo em mente o objetivo de tornar mais confortável a comunicação com objetos mais complexos (usa-se o IMAGE, ao invés de listas intermináveis de parâmetros). É uma forma restrita e controlada de aninhamento de classes.*

- Um conjunto de mensagens (MESSAGES), com ou sem parâmetros, cujos tratadores deverão estar definidos nas XCLASSes que contiverem objetos desta. Estas são necessárias sempre que houver envio de mensagem para o OWNER.

  OWNER <<- M (...);

  ☐ *Em TOOL, um objeto que contém em seu REP objeto de outra XCLASS é dito seu OWNER.*

- Um conjunto de tratadores de eventos assíncronos (HANDLERS), com ou sem parâmetros, que tratam as mensagens recebidas pelo objeto. Estes tratadores podem eventualmente identificar qual o objeto originador da mensagem.

3.2.2 - Métodos

São a comunicação síncrona entre objetos. Admitem parâmetros nos modos IN (valor de entrada), INOUT (entrada e saída) e OUT (valor de saída) à semelhança de ADA [BARNES 82]. Métodos podem ou não retornar valor, podem ter sua definição adiada ("deferred") e podem ser encartados no código como macro instruções (INLINE).

3.2.3 - Mensagens

São o veículo da comunicação assíncrona entre objetos de XCLASSes. Devem ser declaradas sempre que forem enviadas para o OWNER. Constam apenas de um identificador e uma coleção de parâmetros.

3.2.4 - Tratadores de Mensagens (Handlers)

São os tratadores das mensagens enviadas aos objetos. Sua assinatura deve ser igual à da mensagem tratada. Podem ou não, identificar o objeto gerador da mensagem.

Semelhantemente aos métodos, podem também ser adiados(deferred). Por sua característica assíncrona, não podem ser INLINE.

☐ *Exemplo 6.*

```
XCLASS C  -- Classe que trata mensagens
 REP      -- BUTTON_CLICK (de b [k]) e MSG1
   D obj1;
   ARRAY b OF [3] BUTTON ; END ARRAY
 END REP
 HANDLER FOR MSG1 (IN INT a)
   ...
 END HANDLER


 HANDLER FROM b [2] FOR BUTTON_CLICK
   ...
 END HANDLER
 HANDLER FROM b [0], b [1] FOR BUTTON_CLICK
  ...
 END HANDLER
END XCLASS -- C
XCLASS BUTTON -- Classe que envia
 REP            -- BUTTON_CLICK para o OWNER
   BOOL button_on := FALSE;
   ...
 END REP
 MESSAGES
   BUTTON_CLICK;
 END MESSAGES
  ...
 HANDLER FOR MOUSE_CLICK FROM SYSTEM
```

```
XCLASS E -- Classe que envia
   REP    -- MSG1 para C
     C obj;
     ...
   END REP
  METHOD M (...)
     INT I;
     ...
    obj <- NEW ();
    obj <<- MSG1 (I);
       ...
  END METHOD
END XCLASS -- E
```

```
-- SYSTEM é um x_objeto que representa o Sistema Operacional.
   ...
  IF button_on THEN
   button_on := FALSE;
  ELSE
   button_on := TRUE;
  END IF
  SELF <<- REPAINT_BUTTON (); -- SELF é o objeto corrente
  OWNER <<- BUTTON_CLICK;
  RETURN;
 END HANDLER
END XCLASS
```

□ *fim do exemplo 6.*

3.2.5 - Alocação de memória e ponteiros (referências).

Em TOOL, ao declararmos um objeto, podemos acrescentar os seguintes atributos:

- DYNAMIC
- POLY

Objetos "DYNAMIC", "POLY" e objetos de "XCLASSES" (x_objetos) tem sua área alocada dinamicamente no heap. A maneira de alocarmos espaço para um desses objetos é através do método NEW, nativo da linguagem.

□ *Exemplo 7:*

```
XCLASS MAIN
 REP
  DYNAMIC  C1 obj1;
  ARRAY m OF [10]  DYNAMIC INT;
          n OF [10] INT;
  END ARRAY
 END  REP
 METHOD m1 (...);
   ...
   n [1] := 0;
   m [1] <- NEW ();
   m [1] := 0;
 END METHOD
 ...
END XCLASS
```

A utilização de um DYNAMIC difere da de um não "DYNAMIC" apenas na necessidade de criá-lo, pois TOOL prevê a de-referência automática dos ponteiros na utilização

REP, IMAGE bem como a área SHARED são constituídos de objetos. No tocante a REP e SHARED de classe não são permitidas declarações de objetos estáticos da própria classe, o que nos levaria a uma recursão infinita.

As operações com ponteiros podem ser feitas com a utilização de métodos nativos de TOOL e sob algumas retrições. Para que uma referência passe a apontar para um objeto já criado, temos:

- método SAME para objetos DYNAMIC de uma mesma classe.
- método ISNOW para objetos POLY (polimórficos) podendo apontar para objetos de subclasses de sua classe de definição.

☐ *Exemplo 8:*

```
CLASS C from B
 REP
  DYNAMIC C obj1;
  POLY C obj2;
  POLY B obj3;
  DYNAMIC B obj4;
 END REP
 METHOD p (...);
  ...
  obj1 <- NEW ();
  obj2 <- SAME (obj1);
  obj3 <- ISNOW (obj2);
  ...
 END METHOD
END CLASS
```

Neste exemplo vemos:

- A instanciação de um objeto dinâmico obj1 da classe C.
- obj2 passa a apontar para o conteúdo de obj1 (mesma referência).
- obj3 também passa a apontar para o conteúdo de obj2, e neste momento passa a se comportar como objeto definido na classe C.

Para atribuição de valores a objetos, TOOL dispõe de 3 mecanismos:

- Operador de atribuição (:=). Se *obj1 := obj2;* é uma atribuição correta, obj1 e obj2 são da *mesma classe* ou a classe de obj1 é *superclasse da classe de obj2*. Em ambos os casos, o número de bytes copiados é sempre igual ao comprimento do campo receptor, no caso,

obj1. Importante ressaltar que a atribuição **não muda a classe do objeto receptor.**

- método CLONE, que faz a cópia integral da estrutura de um objeto para outro. Este método, quando aplicado a objetos POLY (polimórficos), ***pode eventualmente mudar a classe do objeto receptor.***
- método MCR, que entre objetos de uma mesma família de classes, copia a parte do REP comum às suas classes. Este método também **não muda a classe do objeto receptor.**

☐ *Exemplo 9:*

acrescentando-se P1 à classe C do exemplo 8 teremos:

```
METHOD P1 (...);
  ...
  obj1 <- NEW ();
  obj4 <- NEW ();
  obj4 := obj1;
  obj3 <- CLONE (obj1);
  obj1 <- MCR (obj4);
  ...
END METHOD
```

Vemos neste exemplo:

- a instanciação de obj1 (classe C).
- a instanciação de obj4 (classe B).
- obj4 recebe de obj1 a parte do REP herdada de B.
- obj3 é instanciado e recebe uma cópia integral de obj1, passando a se comportar como objeto da classe C.
- obj1 recebe de obj4 apenas a parte comum entre as classes B e C, que neste caso é o REP de B.

3.2.6 - O compilador.

A linguagem por nós definida e o desejo de simplificar a tarefa de nela se programar nos levou a projetar um compilador de três passos, os quais chamamos **passo 1, passo intermediário e passo 2.**

- O Passo 1 é o responsável pela carga na TS de todos os nomes definidos como *classes, xclasses, métodos, tratadores de mensagens, parâmetros* e componentes dos blocos *REP, IMAGE* e *SHARED*. Nomes ainda não definidos, usados na definição de objetos, são carregados na TS como ***u_classes*** (classes não definidas). Nenhum código é gerado neste passo.
- O passo intermediário verifica se todos os nomes carregados como *u_classes* foram resolvidos (*u_classes são resolvidas quando o passo 1 encontra sua definição*). Nomes ainda não resolvidos são então buscados na **Source_Library** e carregados na TS. A seguir, processa classe a classe, calculando os comprimentos e offsets de cada um dos objetos na TS. Durante este processamento é feita a **verificação de integridade** das relações de herança e composição entre as classes. Esta verificação é feita buscando-se ciclos nas listas de herança e composição.
- O passo 2 faz enfim a análise sintática do corpo dos métodos e tratadores de mensagens e gera o código correspondente.

Esta estratégia "**multipasso**" foi adotada com o intuito principal de simplificar a codificação do programa, não obrigando a declaração antecipada de nomes, cuja definição só aparece após seu uso. Outra razão favorável aos vários passos de compilação é que o cálculo dos endereços dos objetos só pode ser feito após se determinar os tamanhos dos objetos que os antecedem na definição e que podem estar definidos em classes ainda não processadas.

A figura a seguir mostra um esquema da estratégia adotada no compilador para TOOL.

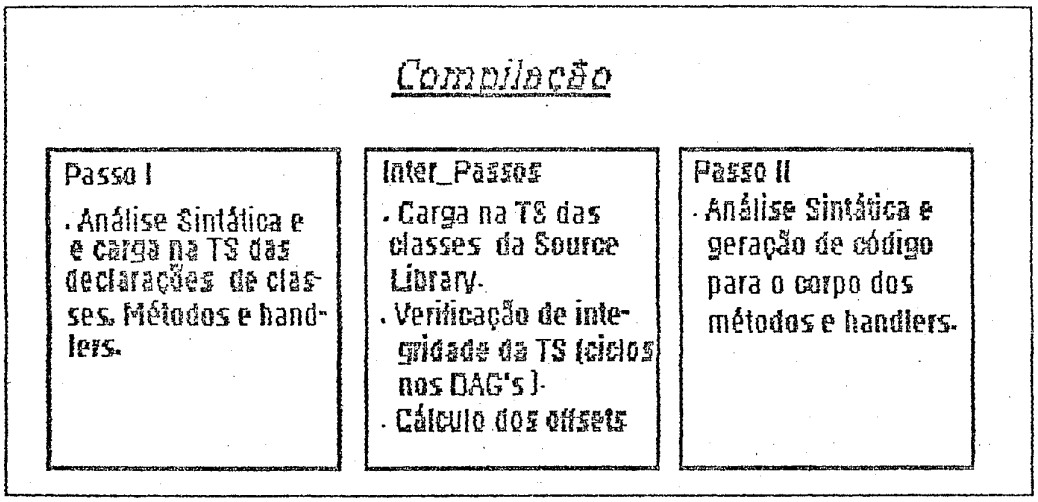


Figura 2. Estratégia adotada no Compilador.

3.3 - Características básicas da TS.

Por tudo que foi exposto, concluímos ser desejável que a TS em questão tenha as seguintes características:

- Dinâmica.

Uma área de tamanho fixo tem dois grandes inconvenientes. Penaliza o pequeno programa alocando mais espaço que o necessário e inviabiliza programas que necessitem de mais espaço. Optamos então por uma alocação de espaço proporcional à necessidade de uso.

- Rápida.

Num levantamento em programas escritos na linguagem TOOL, embora sobre uma amostra pequena, observamos que 50 a 80% dos lexemas (itens léxicos reconhecidos no programa) é composto de identificadores (keywords ou nomes) e portanto acessam a TS.

Outro dado da mesma massa nos dá aproximadamente 3 id's por linha de código, o que nos leva a 1500 acessos à TS em média para um programa de 500 linhas. Optamos pois pelo uso de uma "hash table", que nos deu uma média de 1.17 a 1.25 acessos por nome para um universo de 100 a 400 nomes. Usando-se pesquisa binária, cujo tempo de resposta é da ordem de log(n), temos de 5 a 7 acessos para o mesmo universo.

- Extensível.

[AHO 86] nos diz que poucas linguagens permanecem imutáveis ao longo da vida do compilador. Assim sendo, uma estrutura que permita a agregação de novas informações sem que seja necessário um novo projeto da TS é fundamental. Tratar a TS como classe foi a opção que nos pareceu mais apropriada, pois ao se especializar a classe original podemos acrescentar informações à representação herdada, bem como substituir (redefinir) ou acrescentar métodos aos herdados da classe original.

Optamos por usar quatro tipos de TS's durante a compilação:

- Uma TS agregada ao Analisador Léxico com a finalidade de ajudar no reconhecimento de "keywords".

Uma TS também agregada ao Analisador Léxico com a finalidade de reconhecer e guardar informações sobre as funções "built_in" da linguagem.

- Uma TS chamada global contendo os nomes das classes, seus REP's, Métodos, Handlers e Messages.
- Uma TS chamada local contendo os nomes definidos localmente em Métodos e Handlers.

As três primeiras tem vida global, sendo instanciadas apenas uma vez. Quanto à última, é instanciada e destruída a cada início e final de compilação de método ou handler.

Apesar da linguagem utilizada para implementar o compilador de TOOL não ser orientada a objetos (usamos Microsoft C 5.0), o projeto da TS obedeceu a esta filosofia.

3.3.1 - Representação das TS's.

Todas as TS'S usadas no compilador tem em comum a seguinte definição (o que nos leva à representação da classe TS básica):

- indicação se ativa (bool).
- valor do divisor para a função de Hashing (inteiro).
- "hash_table" (referência para um array de "divisor de hash" elementos, onde cada elemento é uma referência para um símbolo).
- número de símbolos na tabela (inteiro).
- referência para o "pool" de nomes (ver 3.3.11).

As instâncias TS de Keywords, e TS para funções "built_in" são desta classe.

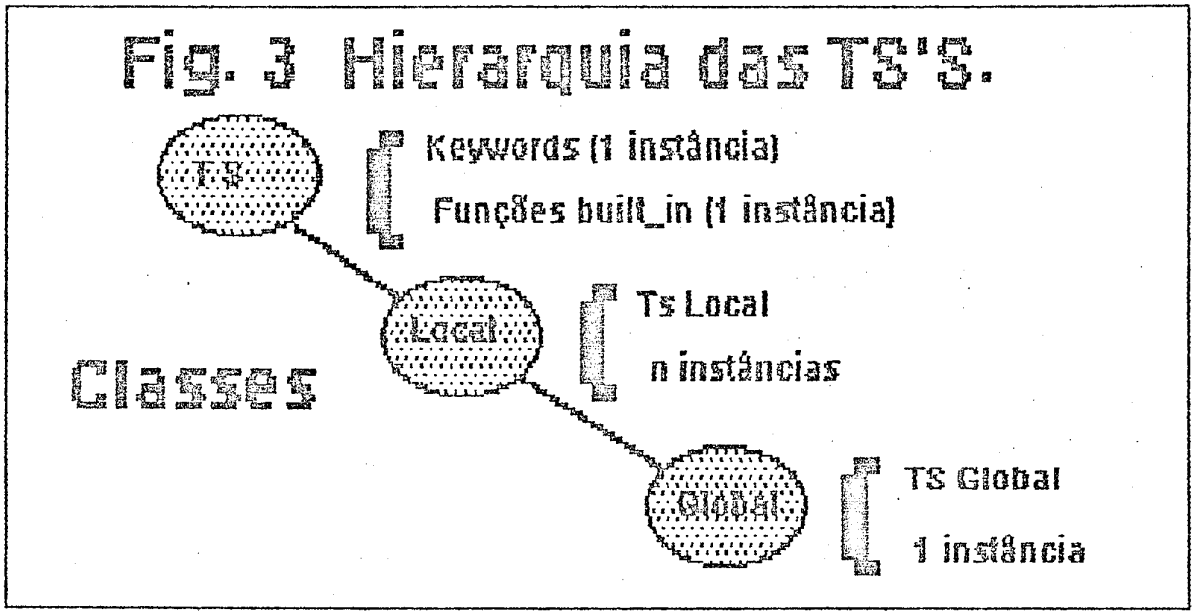
Se acrescentarmos à definição acima:

- referência para o primeiro símbolo.
- referência para o último símbolo.

temos a representação da TS Local, e acrescentando-se a esta última:

- referência para primeira classe no programa (ver 3.3.10).
- referência para última classe no programa.

temos a representação da TS Global.



Em TOOL a representação destas classes fica assim definida:

```
CLASS TS
  REP   -- representação da classe TS básica
    BOOL ativa;   -- indicador se está ativa
    INT divisor_de_hash;
    DYNAMIC ARRAY hash_table OF [divisor_de_hash] POLY SIMBOLO;
    INT num_elems;   -- número de elementos nesta TS
   DYNAMIC POOL_DE_NOMES pool;  -- ref para os nomes dos símbolos
  END REP
END CLASS
CLASS LOCAL_TS FROM TS
  REP   -- representação da TS Local
    POLY SIMBOLO primeiro, ultimo;
  END REP
END CLASS
CLASS GLOBAL_TS FROM LOCAL_TS
  REP   -- representação da TS Global
    DYNAMIC CLASS_REC prim_classe, ult_classe;
  END REP
END CLASS
```

3.3.2 - Métodos das TS's.

A classe TS possui os seguintes métodos:

- **Cria_TS ()** ==> sem parâmetros, aloca espaço para a TS e inicializa os campos do REP.
- **Busca_Simbolo (nome)** ==> busca símbolo cujo nome é igual ao passado como parâmetro. Calcula o "hash code" do nome e faz a busca na lista iniciada em hash_table [hash_code].
- **Guarda_Simbolo (simbolo)** ==> calcula o "hash_code" do nome do símbolo passado como parâmetro, e inclui este no início da lista hash_table [hash_code].

Estes métodos podem ter seus **protótipos** (cabeçalhos de definição) assim definidos em TOOL:

```
METHOD Cria_TS ();
METHOD Busca_Simbolo (IN STRING nome) RETURNS POLY SIMBOLO s;
METHOD Guarda_Simbolo (INOUT POLY SIMBOLO s) RETURNS POLY SIMBOLO
s1;
```

A Classe TS LOCAL possui os seguintes métodos:

- **Busca_Simbolo (nome)** ==> herdado de TS.
- **Guarda_Simbolo (simbolo)** ==> herdado de TS.
- **Cria_TS ()** ==> redefinido. Aloca espaço para esta TS e inicializa os campos do REP.
- **Destroi_TS ()** ==> sem parâmetros. Libera área alocada para esta TS.
- **Liga_Simbolo (simbolo)** ==> inclui simbolo passado como parâmetro na lista iniciada em "primeiro" (ver REP da classe). Estabelece a seguir, as ligações deste símbolo com as listas de dependência entre símbolos (ver Classe SIMBOLO em 3.3.4).

Estes métodos podem ter seus **protótipos** assim definidos em TOOL:

```
METHOD Cria_TS ();
METHOD Destroi_TS ();
METHOD Liga_Simbolo (INOUT POLY SIMBOLO s);
```

A classe TS GLOBAL tem os seguintes métodos:

- **Busca_Simbolo (nome)** ==> herdado de TS.
- **Guarda_Simbolo (simbolo)** ==> herdado de TS.
- **Cria_TS ()** ==> redefinido. Aloca espaço para esta TS e inicializa os campos do REP.
- **Destroi_TS ()** ==> herdado de TS LOCAL.
- **Liga_Simbolo (simbolo)** ==> redefinido. Chama o método de mesmo nome herdado de TS LOCAL e complementa as ligações.

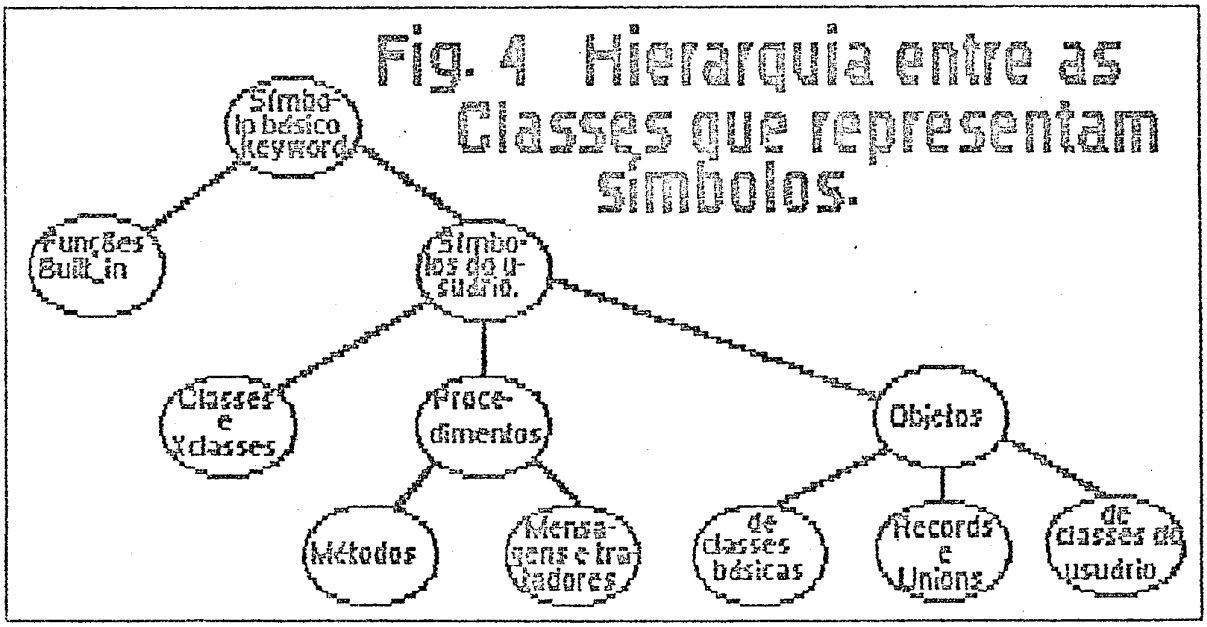
- Traz_Classe_da_Library () ==> Percorre a lista de objetos da classe CLASS_REC e busca na Source Library as classes referenciadas e não definidas. Este método é invocado no Passo Intermediário.
- Verifica_integridade () ==> Percorre a lista de objetos da classe CLASS_REC e verifica se as definições estão consistentes (i.e. se não há definições cíclicas do tipo *A contém B que contém ... que contém A)*.

que em **TOOL** podem ser definidos como se segue:

```
METHOD Cria_TS ();
METHOD Liga_Simbolo (INOUT POLY SIMBOLO s);
METHOD Traz_Classe_da_Library ();
METHOD Verifica_Integridade ();
```

3.3.3 - A representação dos símbolos.

Como podemos observar, as classes definidas acima (TS's) tem por objetivo guardar e recuperar objetos que chamamos símbolos. Para a definição destes, lançamos mão de 11 classes cuja hierarquia pode ser observada na figura seguinte.



3.3.4 - A Classe SÍMBOLO.

A Classe SÍMBOLO da qual derivam todas as demais representações, tem em sua representação:

- referência para o nome.
- token associado (código numérico que indica ao Analisador Sintático a natureza deste símbolo).
- referência para o próximo símbolo de mesmo "hash_code".

Desta classe são os símbolos que representam as palavras reservadas.

3.3.5 - A Classe das funções "built_in" (BUILT_IN).

Esta Classe deriva de SÍMBOLO e acrescenta à sua representação:

- referência para primeiro e último bloco de especificação

Estes blocos de especificação são objetos da classe ESPEC que contem :

- classe do objeto retornado.
- mapeamento da função na Run_Time_Library (RTL). (Neste campo temos o código dado pela RTL à função correspondente).
- número de parâmetros (np).
- array de [np] ocorrências do par: modo e classe.
- referência para a próxima especificação para este símbolo.

☐ *OBS: Consideramos nesta implementação a possibilidade de algumas funções "built_in" serem polimórficas, isto é, poder aceitar mais de uma assinatura. Um exemplo típico deste aspecto é a função ABS, que pode receber como parâmetro um inteiro ou um real e retornar um objeto da mesma classe enviada. Sua primeira especificação mapeia na RTL a função que recebe inteiro e retorna inteiro, enquanto a segunda mapeia a que recebe real e retorna real.*

3.3.6 - A Classe SÍMBOLO_DO_USUÁRIO.

Esta Classe é usada para construir a parte comum dos símbolos definidos pelo usuário. Deriva de SÍMBOLO e acrescenta:

- informação se é visível
- linha e coluna da definição deste nome (para eventual utilização na emissão de mensagem de erro ou na geração de uma referência cruzada).
- comprimento em bytes. Este campo, em princípio, contém o comprimento em bytes necessário para armazenar o objeto representado por este símbolo. Símbolos que necessitam de mais informações a respeito de comprimento estão declarados em classes derivadas desta, com mais informações (este é o caso dos símbolos que representam classes e xclasses).
- referência para o "pai". Este campo é uma referência para o símbolo que lhe é imediatamente superior em hierarquia. Objetos podem ter como "pai":
    - Um RECORD ou uma UNION se são componentes.
    - Uma classe ou xclasse se estão no REP, IMAGE ou SHARED destas.
    - Um método, mensagem ou handler se estão em suas listas de parâmetros.
    - Classes e xclasses têm como "pai" a (x)classes de quem derivam. Métodos, mensagens e handlers têm como "pai" a (x)classe onde estão definidos. Variáveis locais, quando não componentes de RECORDS ou UNIONS, têm como "pai" o método ou handler onde são definidas.
- referência para o "irmão" anterior e posterior. "irmão" significa filho do mesmo "pai".

3.3.7 - Classe SÍMBOLO_DE_CLASSE.

Nesta classe são instanciados os objetos que representam classes e xclasses. Deriva de SÍMBOLO_DO_USUÁRIO e acrescenta a seu REP:

- array de [2] comprimentos herdados (para o REP e IMAGE).
- array de [3] comprimentos (para o REP, IMAGE e SHARED).

- indicador de família. As classes pertencem a famílias. Uma família de classes é o conjunto das classes que descendem de uma mesma classe raiz.
- indicador de nível. Este campo informa a distância (em nós) entre esta classe e a raiz de sua família.
- indicador se há algum método adiado nesta classe (BOOLEAN).
- referência para o objeto da classe CLASS_REC (ver definição em 3.3.10 referente a esta classe).
- array de [3] pares de referências para o primeiro e último "filho" (no REP, IMAGE e SHARED respectivamente). Este conceito de "filho" complementa os conceitos já apresentados de "pai" e "irmão".
- array de [3] pares de referências para o primeiro e último "filho" (método, mensagem ou handler, respectivamente).

3.3.8 - A Classe PROCEDIMENTO.

Esta classe tem originalmente a finalidade de construir a parte comum entre os símbolos que representam métodos, mensagens, e handlers. Dela derivam as classes PROC_METODO e PROC_HANDLER.

Nesta implementação optamos por manter esta classe sem descendência, instanciando nela símbolos que representam métodos, mensagens e handlers.

Esta classe deriva de SÍMBOLO_DO_USUÁRIO e acrescenta ao REP:

- número de parâmetros.
- referência para o primeiro e último parâmetros.
- indicação se é adiado (BOOLEAN).
- indicação se é INLINE (BOOLEAN). Só para métodos.
- classe de retorno. Só para métodos.
- número da mensagem. Só para mensagens.

  ☐ *OBS: Caso o programador deseje se comunicar com algum software "Windows" do qual ele conheça o protocolo de comunicação, pode fixar o número da mensagem, caso contrário, o compilador cria sua própria numeração.*

- referência para classe do objeto que envia a mensagem tratada por este handler. Só para handlers.

3.3.9 - A Classe OBJETOS.

Esta classe, à semelhança da classe anterior (ver 3.3.8), tem originalmente a finalidade de construir a parte comum entre os símbolos que representam objetos. Dela derivam as classes OBJ_BASICO, OBJ_REC_UNION, OBJ_USUÁRIO.

Também neste caso, optamos por manter esta classe sem descendência, instanciando nela símbolos que representam quaisquer objetos.

Esta classe deriva de SÍMBOLO_DO_USUÁRIO e acrescenta ao REP:

- endereço. Endereço é um RECORD contendo o frame e o deslocamento em bytes a partir de seu "pai".

  ☐ *O que chamamos de "frame" são as diversas abstrações de onde um objeto pode estar localizado em nosso modelo de implementação. Frames podem ser global (o objeto está em área de vida global), local (área local na pilha de ativação), objeto (acompanha a localização de seu "pai"), e parâmetro (lista de parâmetros na pilha de ativação).*

- indicação de dependência. (indica a localização da definição do objeto).
- modo (se for parâmetro).
- indicação se é objeto de "heap" ou não.
- indicação de "unsigned". Só para objetos da classe dos inteiros.
- referência para primeiro e último "filhos" (RECORDS e UNIONS).
- indicação se polimórfico. Só para objetos de classes definidas pelo usuário.
- referência para classe. Também só para objetos de classes definidas pelo usuário.

3.3.10 - A Classe CLASS_REC.

Objetos desta classe formam a lista das classes constantes do programa que está sendo compilado. Tem no seu REP:

- Referência para o símbolo (na TS_GLOBAL) que representa esta classe.

- Indicação se já foi definida ou apenas referenciada.
- Referência para próximo objeto desta lista.

Esta lista é toda criada no passo 1 pelo método Cria_Símbolo e usada nos passos 1 e intermediário. No passo 1, ajuda na verificação de duplicidade de nomes na TS e no passo intermediário é usada para se determinar que classes devem ser buscadas na biblioteca.

3.3.11 - A classe POOL_DE_NOMES.

Objetos desta classe formam uma lista e contém os nomes dos objetos constantes de uma TS. Tem em seu REP:

- buffer (2k bytes) para armazenamento dos nomes.
- número de bytes utilizados no buffer.
- referência para o próximo da lista.

> ☐ *Este tipo de armazenamento tem por objetivo minimizar a fragmentação do heap, alocando blocos de 2k bytes para os nomes, ao invés de deixá-los alocados individualmente.*

3.3.12 - As ligações entre os símbolos.

Símbolos estão ligados a outros símbolos através de várias relações funcionais. Estas ligações estão representadas nesta implementação como listas encadeadas. Objetos de outras classes, contendo informações complementares, também estabelecem ligações com símbolos e entre si (ver 3.3.10 CLASS_REC e 3.3.5 BUILT_IN).

O acesso primitivo a um símbolo em qualquer das instâncias de TS é feito através da lista de símbolos de mesmo "hash_code" (hash_table), pelo método Busca_Símbolo definido em SÍMBOLO.

> ☐ *OBS: a definição de hash_table pode ser vista em 3.3.1 (TS).*

As demais listas são:

- Lista de ascendência (para objetos das classes SÍMBOLO e derivadas). Cada símbolo referencia seu "pai".

- Lista de "irmãos" (também para objetos das classes SÍMBOLO e derivadas). Lista duplamente encadeada que aponta para o posterior e anterior "filhos" do mesmo "pai".
- Lista de classes (CLASS_REC). Objetos desta lista referenciam e são referenciados por símbolos da classe SÍMBOLO_DE_CLASSE.
- Lista de especificação (para objetos da classe ESPEC ver 3.3.5). O início desta lista é referenciado por objeto da classe BUILT_IN.

Estas listas estão ligadas entre si através de alguns símbolos. Símbolos de Classes e Xclasses referenciam 6 listas de filhos (objetos no REP, IMAGE e SHARED, métodos, mensagens e handlers). Métodos , mensagens e handlers referenciam a lista de seus parâmetros; RECORD's e UNION's referenciam a lista de seus componentes. Objetos de classes definidas pelo usuário, além das ligações já citadas com "pai" e "irmãos", referenciam sua classe de definição.

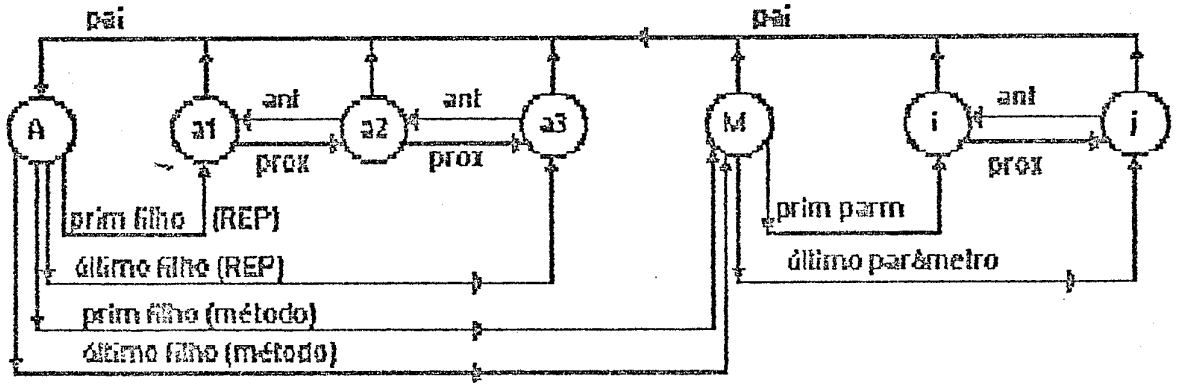
☐ *Exemplo 10.*

Seja a seguinte definição em TOOL:

```
CLASS A
  REP
    INT a1, a2, a3;
  END REP
  METHOD M (IN int i, INOUT REAL j)
    ...
  END METHOD
END CLASS
```

Fig. 5 Ligações na TS para o exemplo 10.

Obs: Não estão mostradas aqui , as listas de símbolos com mesmo hash_code.



3.3.13 - Métodos aplicáveis a símbolos.

Para a construção dos símbolos, todas as classes possuem:

- **Cria_Símbolo ()** ==> aloca espaço e inicializa os componentes.

Para auxílio à Análise Sintática temos:

- **Ascendente (classe)** ==> Verifica se a classe, representada pelo objeto "SÍMBOLO_DE_CLASSE" ao qual o método é aplicado, é ascendente da classe passada como parâmetro. Retorna TRUE ou FALSE. *Este método está definido na classe SÍMBOLO_DE_CLASSE.*
- **Busca_na_hierarquia (nome1, dependência)** ==> verifica se existe um símbolo com nome igual a "nome1" que seja "filho" do símbolo (classe) ao qual o método é aplicado ou que seja "filho" de algum dos símbolos (classes) na lista de ascendência. O parâmetro dependência indica onde procurar (no REP, SHARED, IMAGE, na lista de métodos, mensagens ou tratadores de eventos "handlers". *Este método pertence à classe SÍMBOLO_DE_CLASSE.*

Para auxílio à geração de código temos:

- **Busca_tipo ()** ==> Retorna o tipo (enumeração criada na geração de código) do símbolo ao qual o método é aplicado. *Este método é definido em SÍMBOLO_DO_USUÁRIO.*
- **Processa_tamanho ()** ==> Determina o comprimento em bytes necessário para a alocação do objeto. Este método eventualmente percorre as listas citadas anteriormente. Como é recursivo (para se determinar o comprimento um objeto deve-se determinar os comprimentos de seus componentes), aplicado a uma classe "folha" (i.e. de maior distância da raiz), determina os comprimentos de todo aquele ramo da "família" bem como de todas as demais classes referenciadas. Como sub_produto, calcula os deslocamentos em bytes a partir do "pai". *Este método é "adiado" em SÍMBOLO_DO_USUÁRIO e definido nas classes descendentes desta.*
- **Gera_endereço ()** ==> Cria um objeto da classe NÓ (código intermediário), representando o objeto ao qual o método é aplicado. *Este método é definido na classe SÍMBOLO_DO_USUÁRIO.*

☐ *Obs: Se a estratégia de proteção absoluta dos nomes componentes do REP de uma classe fosse adotada (como em POOL2), uma infinidade de outros métodos deveriam ser criados para atualizar cada um dos campos existentes num símbolo. Nesta implementação, seguimos a estratégia adotada em TOOL que permite acesso aos campos, desde que sejam definidos com o atributo VISIBLE. Desta forma, os métodos de análise sintática podem complementar diretamente as informações sem que, para tanto, tenham que invocar métodos específicos.*

3.3.14 - Considerações sobre a implementação.

Alguns aspectos desta implementação merecem atenção especial:

- Criação automática de classes ainda não definidas. Uma vez que a ordem em que aparecem as classes não deve influir na compilação, sempre que um nome ainda não definido aparece onde se espera um nome de classe, este nome é incluído na TS Global como uma Uclass (undefined class). Caso sua definição apareça posteriormente, suas informações são então atualizadas, existindo para isto métodos específicos. Não sendo definida no programa, o processamento "inter_passos" deve buscá-la na Source Library.
- Símbolos na TS têm a informação sobre sua visibilidade.
- Conforme vimos em 3.2.4, Handlers em TOOL podem especificar o objeto que origina a mensagem a ser tratada. Para estes, apenas para fins de análise sintática, é mantida uma lista com as combinações de mensagem X objeto originador. Esta lista é destruída após o processamento do Passo I.

Na próxima seção apresentamos uma discussão sobre alternativas possíveis e a solução adotada, com comentários sobre as observações feitas acima.

3.4 - Soluções para alternativas.

Ao projetarmos a TS para TOOL, algumas decisões tiveram de ser tomadas.

3.4.1 - Mapeamento das referências na TS.

TS's podem operar sob duas estratégias: utilizando-se nomes únicos ou com múltiplas referências a um mesmo nome. Como já vimos, TOOL admite mesmos nomes desde que em contextos diferentes. Optando-se por múltiplas referências, temos como desvantagens:

- Nem sempre é possível isolar nomes iguais em tabelas diferentes. Isto nos leva a pensar num método de busca que retorne uma lista de símbolos achados ao invés de um único.
- Deixa-se ao analisador sintático a tarefa de decidir qual dos símbolos é o apropriado.

Por outro lado, adotando-se a estratégia de nomes únicos, nos obrigamos a ter sempre presente o contexto sintático atual para que seja feita a prefixação dos nomes, tanto na inclusão como na busca. Optamos por esta estratégia por não sobrecarregar os reconhecedores sintáticos.

O contexto corrente passa então a fazer parte do bloco SHARED da classe. Este contexto é implementado sob a forma de uma pilha de registros contendo o lexema (token), o prefixo para concatenação dos nomes e uma referência para o símbolo (na TS) que o identifica. Outros dados podem ser acrescentados para facilitar o tratador de erros, tais como linha e coluna onde o símbolo aparece no programa fonte.

Optamos também por canalizar todo o acesso às TS's através do método OBTEM_TOKEN (ação, expectativa, classe), da classe ANALISADOR_LÉXICO, que consulta o contexto e decide a formação do nome, tanto na inclusão como na busca.

Exemplificando, o programa:

```
XCLASS X
  REP
    INT a;
    ...
  END REP
  SHARED
    REAL r;
    ...
  END SHARED
  METHOD M (...)
    ...
```

```
  END METHOD
END XCLASS
```

carrega a TS GLOBAL com os seguintes nomes:

- X - Xclass id
- X$a - Objeto da classe dos inteiros, pertence ao REP de X.
- X$r - Objeto da classe dos reais, pertence ao SHARED de X.
- X@M - Método de X.

☐ *Observação: Para eliminar eventuais colisões de nomes, usamos um caracter separador diferente para cada natureza de contexto, única exceção feita aos blocos REP e SHARED, uma vez que TOOL dispensa a prefixação no seu uso.*

3.4.2 Mapeamento dos Handlers na TS

Handlers são mapeados da mesma forma que métodos, com exceção dos que definem o objeto originador da mensagem. Para estes o nome é construído acrescentando-se o nome do originador. Pode-se então ter vários nomes na TS para um mesmo Handler. O primeiro fica na TS durante toda a compilação, os demais são gerados apenas para evitar duplicidade, sendo descartados após o Passo I.

☐ *Exemplo 11.*

a declaração do segundo handler mostrada neste exemplo,

```
XCLASS X
  REP
    BUTTON b1, b2, b3;
  END REP
  ...
  HANDLER FROM b1 FOR BUTTON_CLICK
    ...
  END HANDLER
  HANDLER FROM b3, b1, b2 FOR BUTTON_CLICK
  ~...
  END HANDLER
  ...
END XCLASS
```

deve acusar erro , pois há dois tratadores para o mesmo par mensagem e objeto originador.

4.0 - Complexidade dos algoritmos usados nesta implementação.

A seguir uma breve análise dos algoritmos usados nesta implementação.

4.1 - Cria_TS.

Este método tem apenas um loop de inicialização da hash_table, cujo T = O (n) onde n é o divisor de hash (número de entradas na hash_table).

4.2 - Busca_Simbolo.

Esta busca é sempre feita via hash_table, onde o símbolo é buscado na lista correspondente a seu hash_code. O tempo gasto por este algoritmo é proporcional ao comprimento desta lista. Nesta implementação, para vários universos de 1000 símbolos, a maior lista de colisões teve comprimento 4 e, no pior caso, 85% dos símbolos estava em lista de apenas 1 elemento, o que nos dá:

- melhor caso: T = O (1).
- pior caso: T = O (4).
- caso médio: T = O (1.15).

ou seja, o tempo é constante.

4.3 - Guarda_Simbolo.

Este método também acessa a TS via hash_table. Como a inserção do simbolo é sempre feita no início da lista, o tempo também é sempre constante. T= O (1).

4.4 - Liga_Simbolo.

Este método estabelece as ligações de um símbolo, com seu "pai" e "irmão" antecessor, ou seja, cada vez que este método é invocado, 3 nós são acessados, o que nos leva a um tempo constante. T= O (3).

4.5 - Destroi_TS.

Este método apenas desaloca a área utilizada no heap pela TS e torna sua referência nula. Como a área é alocada em blocos de 2k e há uma limitação de tamanho em 64k, temos T = O (n) n <= 32, onde n é o número de blocos alocados no heap.

4.6 - Traz_Classe da_Library.

Este método percorre uma lista de n classes (CLASS_REC) e traz da Library as classes não definidas. Seja m o número de elementos numa classe da Library e p (p <= m) o número de elementos desta classe que devem ser trazidos. Para a carga de uma classe da Library na TS, chegamos a T = O (m + 4p), pois são:

- m acessos à Library.
- p guardas na TS.
- 3p visitas para ligar os símbolos.
- Seja q o número de classes a serem trazidas da Library (q < n), seja m o número médio de elementos destas classes e seja p o número médio de elementos a serem trazidos destas classes.
- Teremos T = O (n + q x (m + 4p), que, no pior caso (q = n - 1, p = m) nos leva a T = (O (n + (n - 1) x 5m) ==> O (n x m).

4.7 - Verifica Integridade.

Este método, na verdade, se compõe de dois algoritmos distintos:

- Verificação de integridade na ascendência.

- Verificação de integridade na composição.

O primeiro algoritmo percorre todos os nós da lista definida em CLASS_REC e verifica se as listas de ascendência de cada classe não são circulares. Marcando-se os nós já visitados, temos para este algoritmo:

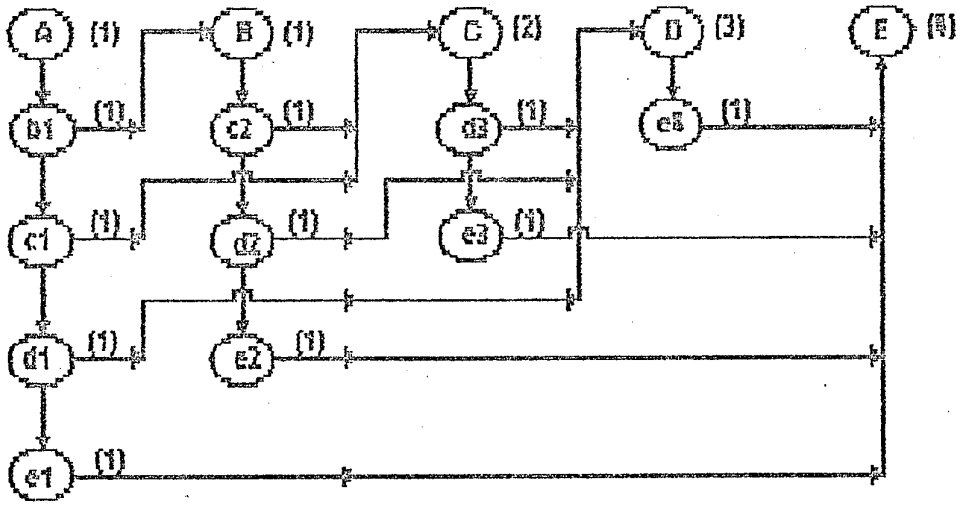
$T = O(2n) ==> O(n)$

☐ *Obs: Cada classe é acessada 2 vezes, uma, via lista de ascendência, e outra, via lista de classes definidas.*

A verificação de integridade na composição é o algoritmo mais "pesado" desta implementação. Deve percorrer, para cada símbolo, todos os componentes de sua classe e assim por diante, até esgotar o grafo ou achar um ciclo.

Este algoritmo tem seu pior caso quando para n classes, cada classe (k) ($k >= 1, k < n$) possui objetos das classes (k+1) a (n).

Fig 6. Pior caso para o "Graph Traversal" na verificação de ciclo.



O número de visitas é:

número de nós no grafo (5 + 4 + 3 + 2 + 1) + acessos extras às classes E, D e C (3 + 2 + 1) respectivamente. Ou seja:

$(n + n - 1 + n - 2 + ... + 1) + (n - 2 + n - 3 + ... + 1)$, que redunda em:

$n (n + 1) / 2 + (n - 2) ((n - 2) + 1) / 2$

enfim, $T = O (n^2 + n + 1) ==> O (n^2)$

4.8 - Cria_Símbolo.

Este método não tem loop, apenas aloca espaço para o símbolo e inicializa seus campos, sendo seu tempo constante. $T = O (1)$.

4.9 - Ascendente.

Este algoritmo percorre uma lista linear (lista de ascendência) a partir de um símbolo classe e verifica se o outro símbolo está nesta lista. Tem seu tempo proporcional ao comprimento desta lista. O caso médio nos dá $T = O (n/2) ==> O (n)$, onde n é o comprimento médio das listas de ascendência de cada classe.

4.10 - Busca_na_hierarquia.

Este algoritmo recebe como parâmetros um símbolo (classe), um nome e um indicador de dependência. Este último serve para o algoritmo determinar em qual das listas do símbolo, a busca deve ser feita. Percorre esta lista sequencialmente buscando um símbolo que tenha aquele nome. Não encontrando, "sobe" na lista de ascendência da classe e repete o processo até achar o nome ou chegar ao fim da lista.

Como podemos observar, o tempo deste algoritmo depende de duas variáveis: o comprimento da lista de ascendência e o comprimento da lista de componentes.

$T = O ((n / 2) (m / 2)) ==> O (n \times m)$, onde n é o comprimento médio das listas de ascendência e m é o comprimento médio das listas de componentes.

4.11 - Busca_Tipo.

Este algoritmo faz uma pesquisa sequencial a uma tabela de "tokens" e retorna o tipo para a geração de código. Em se tratando de pesquisa sequencial, temos T = O (n), onde n é o número de tokens na linguagem.

4.12 - Processa_Tamanho.

Este método usa o mesmo "graph traversal" citado em 4.7 (Verifica_Integridade) e, da mesma forma, tem em seu pior caso T = O (n^2).

4.13 - Gera_Endereço.

Este algoritmo também não possui loop, apenas aloca espaço para o nó inicializando-o com informações obtidas do símbolo. Tem seu tempo constante, T = O (1).

5.0 - Conclusões.

5.1 - O projeto da linguagem e a implemeniação do compilador.

Implementar um compilador é, no mínimo, uma tarefa árdua. Implementá-lo sem uma definição final e completa da linguagem (sintaxe e semântica), torna o processo ainda mais trabalhoso. Isto pode, a princípio, parecer um contra-senso. Há entretanto boas razões para se proceder desta forma, se a linguagem em questão é "nova", isto é, se está em processo de definição. O projeto de uma linguagem de programação, com ambições de colocação nos mundos acadêmico e profissional, é por si só, mais complexo que a implementação de seu compilador.

Diretrizes devem ser tomadas com respeito à forma da linguagem, abrangência, aplicabilidade e paradigma a ser adotado. Deve-se, antes de mais nada, ter bem claros os objetivos desta, isto é, para que universo de aplicações se destina.

Além dos aspectos de corretude, coerência e ortogonalidade, há os mais complicados de se medir, e nem por isso menos importantes, como simplicidade de uso, utilidade e aceitação.

A complexidade de alguns mecanismos definidos na linguagem, só pode ser avaliada quando se projeta o modelo de implementação, pois dependem fundamentalmente do ambiente para o qual se destina o compilador.

> ☐ *Um bom exemplo para este último aspecto pode ser encontrado em TOOL. Mensagens assíncronas entre objetos puderam ser implementadas sem grande esforço adicional porque foi usada a infra-estrutura do Windows 3.0 que já prevê este tipo de serviço.*

Dada a grande rapidez na evolução do estado-da-arte em linguagens de programação e no avanço tecnológico da indústria, esperar-se por uma definição formal e definitiva da linguagem, pode eventualmente, levar o projeto de seu compilador à obsolescência antes mesmo de seu término.

5.2 - A complexidade do projeto de uma TS.

Quanto mais complexa é a semântica de uma linguagem, mais informações são necessárias durante a análise e tradução de cada comando. As extensas dependências entre nomes existentes numa LOO obrigam a que estas informações estejam sempre disponíveis durante as análises sintática e semântica e a geração de código. Isto se reflete diretamente no projeto da TS.

Ao se comparar duas linguagens, podemos analisar a complexidade de seus analisadores sintáticos através de suas gramáticas. Podemos também analisar suas semânticas operacionais ou denotacionais.

Na comparação entre duas TS's podemos, ao invés de analisar suas estruturas, nos ater à complexidade dos algoritmos usados, pois estes são, em princípio, consequência delas.

Uma comparação superficial entre TOOL e a linguagem para o Open Access II (LP_OAII), nos revela que:

- Uma única instância da TS atendeu às necessidades da LP_OAII. TOOL usou 4.
- A estrutura de dados mais complexa na TS da LP_OAII é uma multilista. Em TOOL temos grafos.
- O algoritmo mais "pesado" de TS na LP_OAII é linear (T = O (n)). TOOL, como já vimos na seção 4, tem algoritmos com T = O (n^2).

☐ *A LP_OAII é uma linguagem convencional para acesso a banco de dados de propriedade da SPI (Software Products International - San Diego CA) cujo compilador foi por nós desenvolvido entre 1987 e 1988. Tem como características básicas: Ambientes global e local sem aninhamento. Apenas tipos básicos (inteiros, reais, strings etc) e o tipo data (gregoriana). Comandos de "query" para acesso ao banco de dados, comandos para configuração de janelas e menus, estruturas de controle IF e WHILE e arrays com n dimensões.*

5.3 - Metodologia usada.

O compilador de TOOL foi desenvolvido usando-se basicamente a prototipação. Foi esta a melhor forma encontrada para se levar adiante um projeto do qual muitos aspectos, a princípio, eram desconhecidos da maioria da equipe. Muitos pontos controversos na literatura existente sobre LOO's foram exaustivamente discutidos.

Vale dizer que boa parte do conhecimento que hoje temos sobre orientação a objetos foi adquirido durante e após a fase preliminar da definição de TOOL. Quando o compilador começou a ser implementado, achávamos que a definição da linguagem estava praticamente concluída. Entretanto, boa parte desta foi modificada ao longo do projeto, causando algum transtorno.

A idéia de se usar um modelo orientado a objetos nesta implementação surgiu durante uma das várias reformulações da linguagem, que implicou em mudanças de porte no projeto da TS. Achamos que se fosse possível mapear dentro do compilador entidades que se assemelhassem a classes e implementá-las com esta filosofia, ficaríamos mais a salvo das grandes modificações, pois quase sempre seria possível criar subclasses, herdando o que já existisse, e acrescentar apenas o que houvesse mudado.

A escassa literatura a este respeito nos levou a buscar, entre as metodologias existentes, uma que permitisse um mapeamento simples entre suas entidades e os objetos. A escolha recaiu sobre JSD (Jackson System Development), pois há uma razoável semelhança entre as entidades e os objetos e entre as ações e os métodos.

- ☐ *Entidades e ações são definidas no passo I de JSD [JACKSON - 83].*

É claro que há muitos outros aspectos de orientação a objetos que não estão cobertos em JSD como herança, composição, polimorfismo e tratamento assíncrono de mensagens, mas nos serviu como um bom ponto de partida.

Usando o reconhecimento de entidades (Entity Step) de JSD, conseguimos facilmente identificar as diversas TS's, os vários tipos de símbolos, as unidades de compilação no programa fonte e os muitos tipos de nós do código intermediário.

Um diagrama do relacionamento entre estas entidades nos ajudou na definição das estruturas usando composição.

As interseções existentes entre estas estruturas nos levou a transformar os conjuntos de tipos de TS's e de objetos em famílias de classes ligadas por herança. (ver classes TS e SÍMBOLO no cap 3)

Embora não dispondo de dados numéricos concretos, a impressão que tivemos é que as TS's assim implementadas "suportaram" bem melhor as mudanças que depois surgiram.

Apêndice I

Especificação do pior caso (algoritmo de verificação de ciclo num DAG).

Seja a função herança H(C), (C é um conjunto ordenado de classes) que constrói uma classe C' a partir de C1 .. Cn.

$$C' = H (Ck, Cl, Cm)$$

Uma classe Cj que herda de Ck e Cl e acrescenta objetos ao REP e métodos ao conjunto dos métodos herdados pode ser definida por:

$$Cj = H (Ck, Cl) + L\_REP + L\_MET$$

onde L_REP e L_MET são os acréscimos locais ao REP e ao conjunto de métodos herdados.

O pior caso citado acima acontece quando para um conjunto de classes C = {C1 .. Cn}, temos

$Cn = H (\emptyset) + Lr + Lm$ ($\emptyset$ é o conjunto vazio).

e $Ck = H (Ck+1, .. Cn)$ (para todo k tal que $1 <= k <= n - 1$)

Bibliografia:

- *1 - [AHO - 86] Alfred V. Aho, Ravi Sethi e Jeffrey D. Ullman*
  - Compilers: Principles, Techniques and Tools
  - Addison Wesley
- *2 - [AMERICA - 89] Pierre America*
  - Issues in the Design of a Parallel Object-Oriented Language
  - Formal Aspects of Computing (1989) 1: 366 - 411
  - BCS
- *3 - [BARNES 82] J. G. P. Barnes*
  - Programming in ADA
  - Addison-Wesley
- *4 - [BUHR - 88] P. A. Buhr & C. R. Zarkne*
  - Nesting in an Object Oriented Language is NOT for the Birds
  - ECOOP 88 - pp 128 - 145
- *5 - [CARDELLI 88] Luca Cardelli*
  - A Semantics of Multiple Inheritance
  - Information and Computation n 76 , pp 138 -164 (1988)
- *6 - [GRIES - 71] David Gries*
  - Compiler Construction for Digital Computers
  - John Wiley
- *7 - [GUTTAG - 78] John V. Guttag, Ellis Horowitz e David R. Musser*
  - Abstract Data Types and Software Validation
  - Communications of the ACM dec 78, vol 21 number 12.
- *8 - [JACKSON - 83] Michael A. Jackson*
  - System Development
  - Prentice-Hall

☐ *9 - [MEYER - 89] Bertrand Meyer*

- Object-Oriented Software Construction
- Prentice-Hall

☐ *10- [HOROWITZ - 78] Ellis Horowitz & Sartaj Sahni*

- Fundamental of Computer Algorithms
- Computer Science Press

☐ *11- [MEYER - 89] Bertrand Meyer*

- From Structured Programming to Object-Oriented Design: The Road to Eiffel
- Structured Programming (1989) 1 : 19 - 39
- Springer-Verlag

☐ *12- [SCHAFFERT - 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian & Carrie Wilpolt.*

- An Introduction to Trellis / Owl
- Digital Equipment Corporation, Hudson MA
- Oopsla '86 Proceedings, Sept 86.

☐ *13- [STROUSTRUP - 86] Bjarne Stroustrup*

- The C++ Programming Language
- Addison-Wesley