Projeto e Implementação de $\mathcal{LL}$: Uma Linguagem de Bibliotecas Baseada em Objetos

Evandro Bacarin

Projeto e Implementação de $\mathcal{LL}$: Uma Linguagem de Bibliotecas Baseada em Objetos

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Evandro Bacarin e aprovada pela Comissão Julgadora.

Campinas, 17 de fevereiro de 1995.

Prof. Tomasz Kowaltowski
Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Projeto e Implementação de $\mathcal{LL}$: Uma Linguagem de Bibliotecas Baseada em Objetos [1]

Evandro Bacarin[2]

Departamento de Ciência da Computação
IMECC – UNICAMP

Banca Examinadora:

- Hans Kurt Edmund Liesenberg[3]
- Luiz Eduardo Buzato (Suplente)[3]
- Luiz Henrique de Figueiredo[4]
- Tomasz Kowaltowski (Orientador)[3]

---

[1] Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação da UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Este trabalho foi desenvolvido com apoio financeiro do Conselho Nacional de Pesquisa e Desenvolvimento Tecnológico (CNPq) e da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP).

[2] O autor é Bacharel em Ciência da Computação pela Universidade Federal de São Carlos.

[4] Pesquisador do CNPq associado ao TeCGraf, Departamento de Informática, PUC-Rio.

[3] Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

Aos meus pais

Ao Claudinho

Agradecimentos

Estas, talvez, sejam as linhas mais difíceis de serem postas no papel. Foram tantas as pessoas que, direta ou indiretamente, contribuíram para que este trabalho se concluísse. Se ousasse citá-las todas, não escaparia da armadilha do esquecimento.

Seria injusto, entretanto, se não lembrasse das pessoas mais diretamente envolvidas no trabalho: o prof. Tomasz, pela orientação segura e pela dedicação; o Apuã, o Clevan e a Cris, pelo auxílio. A primeira idéia da linguagem foi sugerida pelo prof. Pedro J. Rezende e foi enriquecida pelas sugestões do prof. Jorge Stolfi.

Não poderia esquecer do incentivo e da confiança dos professores da Federal, sobretudo do Zorzo e do Durval.

Os esforços do setor administrativo do DCC foram, também, relevantes.

É, neste momento, impossível agradecer a todos os amigos, amigas ou colegas (foram muitos!) pelo incentivo, apoio e companheirismo que jamais foram escassos neste três anos. Não cito qualquer nome, pois não resistiria à tentação de tentar rememorá-los todos. Por certo omitiria vários dos mais caros.

Resumo

O trabalho apresenta uma linguagem de programação, denominada $\mathcal{LL}$ (de $\mathcal{L}$ibrary $\mathcal{L}$anguage), que visa atender as necessidades de projetistas de algoritmos — programadores ocasionais que necessitam implementar seus algoritmos para testá-los, demonstrá-los didaticamente ou analisar sua eficiência.

A linguagem busca ser simples e expressiva. Uma linguagem de programação complexa, em contraste com a simplicidade perseguida no projeto de $\mathcal{LL}$, pode induzir o programador a erros alheios ao algoritmo em implementação, devido à complexidade excessiva da ferramenta.

A expressividade é outra qualidade desejável, uma vez que os algoritmos a serem codificados são, muitas vezes, descritos em artigos, livros ou anais de congressos em uma linguagem de bastante alto nível.

Expressividade e simplicidade visam prototipação rápida de algoritmos. O tempo e o esforço de implementação podem ser abreviados se o programador tiver ao seu alcance um rico conjunto de bibliotecas que o auxiliem em seu trabalho. Assim, $\mathcal{LL}$ enfatiza a facilidade de criação e reutilização de bibliotecas escritas em outras linguagens.

O trabalho descreve, também, alguns detalhes de implementação de seu compilador, bem como relaciona semelhanças e diferenças de linguagens de programação que influenciaram seu projeto.

Abstract

This thesis introduces a programming language called $\mathcal{LL}$ (for Library Language). Its main goal is to answer needs of algorithm designers who as occasional programmers implement their algorithms in order to test them, to demonstrate them in a didactic way or to analyze their efficiency.

$\mathcal{LL}$ is simple and expressive. A complex programming language can lead the programmer to errors related not to the algorithms that are being implemented, but to the excessive complexity of the tool being used.

Another goal to be achieved is expressiveness, since the goal is to implement algorithms which are generally shown in scientific papers, books or proceedings, in a high level language.

Expressiveness and simplicity lead to rapid prototyping of the algorithms. Time and effort in implementation can be reduced if the programmer has available a rich variety of libraries. Thus $\mathcal{LL}$ emphasizes ease in the creation and reuse of libraries written in other languages.

Some of the compiler implementation details are also shown as well as similarities and differences with the programming languages which influenced the design.

Conteúdo

1 Introdução 1

2 Discussão sobre a Linguagem 5
- 2.1 Aspectos léxicos e sintáticos . . . 5
- 2.2 Programa *LL* . . . 6
- 2.3 Objetos . . . 7
- 2.4 Declaração de procedimentos e funções . . . 12
- 2.5 Variáveis, constantes e parâmetros formais . . . 14
- 2.6 Expressões . . . 16
- 2.7 Comandos . . . 17
  - 2.7.1 Atribuição . . . 17
  - 2.7.2 Comando condicional . . . 17
  - 2.7.3 Comando de seleção . . . 18
  - 2.7.4 Comandos iterativos . . . 19
  - 2.7.5 Comandos de interrupção . . . 20
  - 2.7.6 Invocação de procedimento . . . 20
- 2.8 Extensões sintáticas . . . 21
- 2.9 Módulo *System* . . . 22
  - 2.9.1 Pseudofunções . . . 23
  - 2.9.2 Exemplo de um módulo System típico . . . 23

3 Comparações com outras Linguagens 29
- 3.1 Controle de Visibilidade . . . 30
- 3.2 Estruturação dos dados . . . 31
- 3.3 Estruturas de Controle . . . 32

4 Implementação do Compilador 36
- 4.1 Estruturas de Dados . . . 37

4.1.1 Tabela de Símbolos 37
4.1.2 Árvore de Programa 38
4.2 Processo de Compilação 41
4.3 Arquivos Intermediários 43
4.4 Sistema de Execução 44
4.4.1 Pilha de Execução 45
4.4.2 Seleção dinâmica 47
4.5 Geração de Código 48
4.5.1 Cláusulas de importação 51
4.5.2 Cláusulas de exportação 51
4.5.3 Constantes 51
4.5.4 Tipos 52
4.5.5 Procedimento e Funções 54
4.5.6 Pseudofunções 56
4.5.7 Corpo do módulo 58
4.5.8 Comando forall 58

5 Exemplo **61**
5.1 Algoritmo 61
5.2 Implementações 62
5.3 Comparação 63

6 Conclusões e trabalhos futuros **68**

Bibliografia **70**

Lista de Figuras

2.1 módulo $\mathcal{LL}$. 7
2.2 tipo $\mathcal{LL}$. 8
2.3 hierarquia de tipos. 9
2.4 inserção do operador @. 10
2.5 tipos Ponto e Segmento. 10
2.6 tipo NovoPonto. 11
2.7 declaração de rotinas. 12
2.8 cadeia de implementação. 13
2.9 comando **case**. 14
2.10 declaração de variáveis. 15
2.11 declaração de constantes. 15
2.12 implementações externas 28

4.1 tabela de símbolos 37
4.2 árvore de programa 40
4.3 compilação de m.ll 44
4.4 seleção do tipo. 48
4.5 seleção do método. 49
4.6 arquivo m.lls. 50

5.1 polígono e ponto interno. 62
5.2 polígono de visibilidade. 63
5.3 polígono de visibilidade é encoberto. 64
5.4 aresta encoberta pelo polígono de visibilidade. 65
5.5 algoritmo. 65
5.6 versão $\mathcal{LL}$. 66
5.7 versão C++. 67

Capítulo 1

Introdução

O trabalho proposto consiste do projeto de uma linguagem de programação, denominada $\mathcal{LL}$,[1] e implementação de seu compilador.

A principal motivação para o projeto desta linguagem foi encontrada em algoritmos envolvendo estruturas de dados complexas, descritos em livros, anais de congressos, periódicos especializados, relatórios técnicos e, em particular, a implementação de algoritmos geométricos no ambiente `Geolab` [Jac] com a utilização da biblioteca `Leda` [Nah]. Em geral, o algoritmo é extraído de sua prova de correção. Algumas dessas provas, por exemplo, fazem uma análise detalhada e exaustiva das possíveis situações que podem surgir durante a resolução do problema. A algoritmização da prova é quase direta. O passo seguinte, a implementação, entretanto, é bastante largo. Vários problemas podem emergir durante o percurso deste degrau.

O primeiro deles é a forma dos dados manipulados, em geral, bastante diversa das estruturas primitivas de linguagens de programação. Em lugar de valores inteiros, caracteres, valores lógicos, etc, o programa manipulará pontos, retas, nós, arestas de grafos, entre outros. Este problema é facilmente resolvido através da utilização de construtores de tipos, como registros, matrizes.

Um segundo problema é o que se costuma denominar "hiato semântico" entre a descrição do algoritmo e sua implementação. Muitas vezes a linguagem escolhida não é tão poderosa quanto se deseja, fazendo com que seja difícil identificar o algoritmo como esqueleto do programa, dificultando sua compreensão.

Por fim, muitas linguagens de programação, por serem complexas em demasia, por apresentarem muitas particularidades, induzem erros de programação alheios ao algoritmo em implementação e exigem um alto grau de especialização

[1] Library Language

do programador para que sejam usadas corretamente.

A linguagem $\mathcal{LL}$ tenta resolver os últimos dois problemas. Seu público-alvo são projetistas de algoritmos — programadores ocasionais que necessitam implementar seus algoritmos para testá-los, demonstrá-los em um nível didático e analisá-los em relação a sua eficiência.

Estes pontos implicam que uma linguagem adequada a este grupo de programadores deve ser simples, não esconder gastos de tempo e memória não constantes, porém, deve ser bastante expressiva. Simplicidade, neste contexto, significa que a linguagem possua poucos conceitos de programação que devem ser dominados pelo programador, e que os próprios conceitos sejam, também, claros. O conceito de entrada e saída, por exemplo, não foi incluído na linguagem, pois pode ser facilmente inserido em sua biblioteca padrão.

Em vista disso, $\mathcal{LL}$ apresenta estruturas sintáticas poderosas e legíveis e que tentam evitar manipulações não seguras dos dados. Não são previstas, por exemplo, conversões irrestritas de tipo, como as encontradas na linguagem C.

Deseja-se, também, e sobretudo, incentivar farta utilização de bibliotecas, induzindo a uma alta reusabilidade de código. Para que esta reusabilidade seja efetiva, $\mathcal{LL}$ provê mecanismos que permitem a utilização de bibliotecas escritas em outras linguagens. Para que estes mecanismos sejam eficientes e de fácil compreensão, $\mathcal{LL}$ é uma linguagem baseada em objetos. É importante frisar que $\mathcal{LL}$ não possui qualquer tipo predefinido. Os principais tipos (que em outras linguagens seriam considerados primitivos) devem ser implementados de forma semelhante a qualquer tipo definido pelo programador.

Deve ser notado que os termos "orientação a objetos" ou "baseado em objetos" não possuem consenso quanto a seus significados. Cardelli e Wegner [CW85] e Wegner [Weg87] afirmam que uma linguagem é orientada a objetos se: suporta objetos constituídos por um estado interno e por uma interface de operações que os manipulam; objetos pertencem a classes e, finalmente, hierarquias de classes podem ser definidas através de herança, o que permite que sub-tipos herdem atributos de suas super-classes.

Uma diferença fundamental, entretanto, pode ser verificada nas duas abordagens. Para Cardelli, objetos são encapsulados, isto é, seu estado interno é acessível apenas pelas operações de suas interfaces. Wegner, por sua vez, apenas exige que objetos possuam um estado interno que possa influenciar ou ser influenciado pela aplicação de operações de suas interfaces. Tal abordagem causa algumas dúvidas em respeito a sua utilidade, uma vez que uma das principais motivações para a orientação a objetos foi a reutilização de código. A ausência de encapsulamento pode, como bem sabido, dificultar esta reutilização.

Blair e outros [B+89], por outro lado, sustentam que a principal restrição na

definição de Wegner se deve ao fato de ser baseada em mecanismos em lugar de propriedades, isto é, põe em evidência detalhes de implementação da linguagem de programação em lugar de enfatizar as propriedades que o sistema deve oferecer ao usuário. Eles propõem que uma linguagem (ou mais genericamente, qualquer sistema) para ser considerada orientada a objetos deve apresentar as seguintes propriedades:

- encapsulamento: dados e operações devem estar associados em uma única entidade (objeto); além disso, o acesso a objetos deve ser feito unicamente através de uma interface de operações bem definida;
- abstração baseada em conjuntos: todas entidades são consideradas pertencentes a conjuntos (não necessariamente disjuntos) que abstraem propriedades e comportamentos comuns;
- polimorfismo de inclusão: como objetos podem pertencer a mais de um conjunto simultaneamente, operações aplicáveis a objetos de um dos conjuntos devem ser aplicáveis a objetos dos outros conjuntos. Isto permite que objetos possam se comportar adequadamente em contextos diferentes ou que objetos compartilhem comportamentos (reutilização de código);
- polimorfismo de operação: operações de um mesmo nome devem poder ser aplicadas a diferentes objetos que não possuem nenhum relacionamento de inclusão.

$\mathcal{LL}$ poderia ser considerada orientada a objetos de qualquer um dos pontos de vista, porém, não requisita tal título uma vez que, desde sua gênese, não tinha por ambição ser enquadrada neste paradigma. Uma das premissas de seu projeto era estar aberta a inclusão de mecanismos (orientados a objetos ou não) que a auxiliassem ir de encontro a seus objetivos.

A fim de que se concilie simplicidade e expressividade, $\mathcal{LL}$ apresenta dois níveis de sintaxe. O primeiro, a *sintaxe estrita*, possui o essencial em termos de construções sintáticas, possibilitando a implementação dos conceitos fundamentais da linguagem.

O segundo nível, a *sintaxe estendida*, apresenta construções que tentam deixar o código mais compacto e, sobre tudo, mais legível. Muitas vezes exige a existência de certos métodos, ou apenas, apresenta uma forma mais agradável para certas construções.

Quanto à organização da linguagem, $\mathcal{LL}$ é estruturada em quatro níveis: (1) um programa $\mathcal{LL}$ é descrito em um módulo e pode fazer uso de sub-rotinas,

constantes e tipos descritos em outros módulos; (2) tipos e suas operações (métodos) podem ser implementados em alguma linguagem de suporte (`C++`, por exemplo), com o auxílio de (3) bibliotecas desta linguagem e serem feitos conhecidos ao programa $\mathcal{LL}$ através de (4) uma interface.

O próximo capítulo descreve a linguagem $\mathcal{LL}$, seus aspectos sintáticos e semânticos, discute algumas decisões de projeto e apresenta algumas possíveis atualizações.

O terceiro capítulo discute a relação da sintaxe e semântica de $\mathcal{LL}$ com as de outras linguagens de programação, tais como: `C++`[Str86], `Trellis`[S+85], `Modula-2` [Wir82], `Modula-3` [Nel91] e `Smalltalk` [GR83]. São tratadas questões como mecanismo de controle de visibilidade, herança, iteradores, polimorfismo, reutilização de bibliotecas.

O capítulo 4 aborda tópicos sobre a implementação do compilador como, por exemplo, estruturas de dados internas, processo de compilação, sistema de execução da linguagem e geração de código.

O capítulo 5 inclui um exemplo de utilizacao da linguagem $\mathcal{LL}$.

Por fim são apresentadas algumas conclusões e contribuições do trabalho e são delineadas sugestões para trabalhos futuros.

Capítulo 2

Discussão sobre a Linguagem

Nas próximas seções são discutidas características de $\mathcal{LL}$ e são apresentadas justificativas sobre algumas decisões de projeto. O capítulo, entretanto, não pretende ser um manual de referência da linguagem, outrossim, apresenta a sintaxe e a semântica de $\mathcal{LL}$ informalmente através de alguns exemplos. A descrição completa da linguagem pode ser encontrada em [KB93].

A convenção usada em trechos de programas $\mathcal{LL}$, ou em descrições de sua sintaxe, é sumarizada a seguir:

- palavras em negrito (p.ex., **module**, **type**) são reservadas;
- textos entre [ ] são opcionais;
- o símbolo "$\leftarrow$" é utilizado em lugar de :=; "$\Leftarrow$", em lugar de ::=; "$\neq$", para <> e "$\Rightarrow$" para =>.

2.1 Aspectos léxicos e sintáticos

Como na maioria das linguagens de programação, um programa $\mathcal{LL}$ é lexicamente composto por palavras-reservadas, identificadores, denotações, operadores, símbolos especiais, cadeias de implementação, pragmas (dependentes da implementação) e comentários (de linha, iniciado por ||, e de bloco *(* *)*).

Palavras-reservadas, ao contrário de *identificadores*, não são distingüíveis pela utilização de letras maiúsculas ou minúsculas. Assim, *type*, *Type* ou *TYPE* referem-se à mesma palavra-reservada (**type**), enquanto que *nome*, *Nome* e *NOME* são identificadores distintos. *Denotações*, por sua vez, podem ser *inteiras* (*1994*), *reais* (*3.14*, *8.06e-15*, *1.*), *cadeias de caracteres* ("*batatinha quando nasce ...*") ou

coleções({*1,2,"caju",X*}, <<*F1,F2,"umbu"*>>, [[*x+y, 2.15*]]). Embora reconhecidas, os significados das denotações não são definidos por $\mathcal{LL}$, mas determinados em tempo de execução, pela invocação de *pseudofunções* adequadas (seção 2.9.1).

A linguagem $\mathcal{LL}$ possui um rico conjunto de *operadores* (pré-fixos, infixos e pós-fixos), entretanto não determina seus significados. Outrossim, a aplicação de um operador é tratada como uma invocação a um método (seção 2.6). *Cadeias de implementação* (*'ImplProc1();'*) são seqüências de comandos na linguagem hospedeira, não interpretados pelo compilador $\mathcal{LL}$, e podem ser usadas somente em módulos de ligação (interfaces).

Seguindo a tradição de Pascal [JW88], $\mathcal{LL}$ se utiliza do conceito de blocos de comandos, cada um dos quais encerrado por um terminador adequado (**endif**, para comandos condicionais; **endcase**, para comandos de seleção, etc). Por vezes, num aninhamento relativamente profundo, vários blocos de comando encerram-se simultaneamente, provocando um acúmulo de terminadores num único ponto. Alternativamente, poderia ser dado significado sintático à indentação do programa em substituição aos terminadores. Esta alternativa, entretanto, não foi adotada uma vez que tal "transtorno" não parece tão crítico a ponto de se impor ao programador um estilo de codificação.

Visando aumentar a legibilidade de seu código, $\mathcal{LL}$ apresenta algumas construções sintáticas (*sintaxe estendida*, seção 2.8), que são automaticamente substituídas por invocações a métodos adequados. Por exemplo:

$$A \leftarrow M[x,y,z]$$
$$M[x,y,z] \leftarrow E$$

são respectivamente interpretados como:

$$A \leftarrow Value(M,x,y,z)$$
$$Update\ (M,x,y,z,E)$$

2.2 Programa $\mathcal{LL}$

Um programa $\mathcal{LL}$ é constituído por um módulo principal e por um conjunto (possivelmente vazio) de outros módulos dos quais importa declarações.

Um módulo compreende uma *interface*, onde são especificados nomes importados e exportados pelo módulo; um conjunto de *declarações* (constantes, tipos, procedimentos e funções) e um corpo executável (figura 2.1). Comandos são utilizados em corpos de sub-rotinas: construtores, métodos, rotinas (procedimentos e funções) e no corpo do módulo; criam e manipulam objetos. Alternativamente, o corpo de uma sub-rotina pode ser substituído por uma cadeia de implementação.

```
module Exemplo is

   import modulo1;
   from modulo2 import nome1, nome2;

   const PI is 3.1415;

   type Ti is
      ...
   endtype;

   procedure P (X;Y;Z) is
      (* comandos *)
   endprocedure;

   function F (X) is
      (* comandos *)
   endfunction;

begin
   (* comandos *)
endmodule
```

Figura 2.1: módulo $\mathcal{LL}$.

2.3 Objetos

Objetos são instâncias de tipos (figura 2.2), que definem construtores e as operações a eles aplicáveis (métodos). Possuem uma marca identificando seu tipo. Métodos podem ser procedimentais ou funcionais (incluindo operadores unários e binários).

Nomes de métodos podem ser sobrecarregados, isto é, um método m pode possuir várias implementações, a menos que duas delas sejam declaradas no mesmo tipo, sejam simultaneamente procedimentais (ou funcionais) e possuam a mesma aridade. Implementações distintas de construtores que sobrecarregam um mesmo nome são distingüidas por seu tipo e sua aridade.

Construtores podem ou não ter parâmetros formais, enquanto que métodos devem possuir ao menos um parâmetro. O primeiro argumento de uma invocação a um método identifica o objeto sobre o qual o método terá efeito.

```
type Vector is

   constructors
      |New (Tam: inteiro)          ⇒ 'implNew();'
   functions
      |Valor (V,I: inteiro)        ⇒ 'implValor();'
      |Tamanho(V): inteiro         ⇒ 'implTamanho();'
      |Maximo(V)                   ⇒ Maior := Valor(V, Tamanho(V));
                                      forall V in 1..Tamanho(V) - 1 do
                                         if Valor(V,I) > Maior
                                            then Maior := Valor(V,I)
                                         endif
                                      endforall;
                                      return Maior;
   operators
      |(#V): inteiro               ⇒ return Tamanho(V);
      |V!                          ⇒ return Maximo(V);
      |(V1 * V2 <:Vetor): Vetor    ⇒ 'implProdVetorial();'
   procedures
      |Inicie(V,valor)             ⇒ 'implInicie();'
endtype;
```

Figura 2.2: tipo $\mathcal{LL}$.

Parâmetros formais e resultados de métodos-funcionais podem apresentar restrições de tipos sobre os argumentos da invocação ou sobre o valor de retorno, verificadas em tempo de execução. As restrições podem ser $:T$, significando que o objeto deve ser do tipo T, ou $<:T$, impondo que o objeto seja do tipo T ou um de seus sub-tipos.

A implementação de métodos e construtores pode ser feita através de comandos $\mathcal{LL}$ ou por cadeias de implementação.

$\mathcal{LL}$ não possui qualquer tipo pré-definido. Todos os tipos devem ser implementados em módulos, utilizando os recursos da linguagem. Entretanto, supõe que os tipos *root*, *boolean*, *typetag*, *undefined*, a constante *true*, do tipo *boolean*,[1] sem o que não seria possível a avaliação de expressões condicionais ou que envolvam tipos de objetos, e a constante *UndefinedValue* do tipo *undefined*, são declarados no módulo *System* (seção 2.9).

[1]Isto não implica que sejam pré-definidos. $\mathcal{LL}$ não conhece suas implementações, apenas solicita que tais nomes existam.

A declaração de um tipo provoca a criação de uma constante (com o mesmo nome) do tipo *typetag* visível apenas em seu módulo (salvo se exportada). Esta constante possibilita a comparação entre tipos de objetos, permitindo, por exemplo, a construção de rotinas polimórficas.

Um tipo S pode ser declarado como sub-tipo de um tipo T (figura 2.3), definindo uma relação hierárquica entre tipos de um programa $\mathcal{LL}$. Caso o super-tipo não seja explicitamente especificado, S é suposto ser sub-tipo de *root*.

type S **subtypeof** T **is**
...
endtype

Figura 2.3: hierarquia de tipos.

Os métodos declarados por T são automaticamente herdados por S (a menos que sejam redefinidos).

Há de se notar que a declaração de tipos não prevê a possibilidade de definição de campos de dados comumente encontrados em linguagens de programação tradicionais (por exemplo, registros em Pascal, classes em C++). Tal característica foi deliberadamente excluída da definição da linguagem a fim de simplificar a tarefa de programação de interfaces[2] para $\mathcal{LL}$. Note-se que um programador de interfaces necessita conhecer a estrutura de um objeto $\mathcal{LL}$ e do sistema de execução da linguagem para desempenhar adequadamente sua tarefa. A inclusão de tal característica tornaria a estrutura dos objetos mais complexa, principalmente se campos pudessem ser herdados e redefinidos dentro da hierarquia de tipos.

É importante, ainda, ressaltar que $\mathcal{LL}$ não visa prover mecanismos sofisticados para construção de tipos, outrossim, tem por objetivo fazer bom uso de estruturas de dados criados por bibliotecas.

Entretanto, o modelo de herança de tipos de $\mathcal{LL}$ tal como correntemente definido, apresenta duas deficiências. A *primeira* está associada ao fato de se supor que a representação interna de sub-tipos seja a mesma de seus super-tipos, tornando confiável a operação de *recast*.[3] Porém, permite-se que sejam utilizadas cadeias de implementação em qualquer nível da hierarquia de tipos a fim de se implementar construtores. Considere, por exemplo, a inclusão da operação *nand*, representada pelo operador @, ao tipo *boolean* (figura 2.4).

[2] Interfaces entre bibliotecas externas e $\mathcal{LL}$.

[3] O objeto assume o tipo de seu super-tipo imediato.

```
type myBoolean subtypeof boolean is

   constructors
      |myTrue ()              ⇒ ...
      |myFalse ()             ⇒ ...
   operators
      |(x @ y) <: myBoolean ⇒ return not (x and y)
endtype;

const myTrue is myBoolean.myTrue();
const myFalse is myBoolean.myFalse();
```

Figura 2.4: inserção do operador @.

É necessária a criação de objetos do tipo *myBoolean* (constantes *myTrue* e *myFalse*, por exemplo) a fim do operador @ ser aplicável. Porém, se os construtores de *myBoolean* forem implementados externamente, caberá ao programador de interface garantir a compatibilidade entre as representações internas de *boolean* e *myBoolean*, assegurando o perfeito funcionamento de métodos herdados.

A *segunda deficiência* é relacionada com a possibilidade de vários tipos serem baseados em uma mesma implementação externa através da herança de tipos. Por exemplo, sejam as classes `Point` e `Segment` escritas em `C++`. Um objeto da classe `Segment` é definido por dois objetos da classe `Point`. Esta classes podem ser mapeadas ao tipos *LL Ponto* e *Segmento* (figura 2.5).

```
type Ponto is
   ...
endtype;

type Segmento is
   constructors
      |New (p1,p2 <: Ponto)   ⇒ ...
   functions
      |P1 (S) <: Ponto        ⇒ (* retorna p1 *)
endtype
```

Figura 2.5: tipos Ponto e Segmento.

Caso um programador $\mathcal{LL}$ desejasse acrescentar um novo método ao tipo *Ponto*, seria necessária sua especialização (figura 2.6):

```
type NovoPonto subtypeof Ponto is
   constructors
      |New (x,y)      ⇒ ...
   procedures
      |M (p)          ⇒ ...
endtype
```

Figura 2.6: tipo NovoPonto.

Assim, seria permitido criar-se segmentos a partir de instâncias de *Novo-Ponto*. Porém, a representação interna de uma instância de segmento é baseada na classe `Segment`. Esta, por sua, vez utiliza instâncias da classe `Point`. Salvo "manobras" na implementação, a representação interna de um objeto do tipo *Segmento* não distingüiria os tipos dos pontos que o compõe. O método $Segmento.P_1(S)$ apenas retornaria objetos do tipo *Ponto*. Assim,

$$M(P_1(Segmento.New(NovoPonto.New(1,1),NovoPonto.New(10,10))))$$

não é aplicável.

Os problemas descritos ocorrem basicamente pela tentativa de implementar dois conceitos distintos através de um único mecanismo da linguagem. Primeiro, $\mathcal{LL}$ não predefine qualquer tipo por motivos já expostos e, por isso, necessita de um mecanismo que permita a inclusão de implementações externas à linguagem. Segundo, o conceito de especialização de tipos através de sub-tipagem é sabido ser um mecanismo adequado para reutilização de código, propiciando simplicidade e clareza a um programa. Este é, portanto, um conceito aderente à filosofia de $\mathcal{LL}$.

Estes dois conceitos, entretanto, foram implementados através do mecanismo de declaração de tipos. A solução natural é criar mecanismos distintos que implementem cada um dos conceitos. A especialização poderia continuar a ser feita a partir da definição de tipos, porém seus construtores e métodos não poderiam utilizar cadeias de implementação. Seriam introduzidos "tipos de ligação", semelhantes aos primeiros, porém, não possuiriam super-tipos e cadeias de implementação apenas poderiam ser usadas para implementar seus construtores ou métodos.

A segunda deficiência, entretanto, não é resolvida, pois não foram criados mecanismos que permitissem mapear um tipo (classe) da linguagem hospedeira

a dois ou mais tipos $\mathcal{LL}$, isto é, relacionar, por exemplo, a classe `Point` aos tipos *Ponto* e *NovoPonto*, e poder distinguir qual dos dois mapeamentos se verifica em um dado instante.

A inclusão de tal mecanismo à linguagem muito provavelmente acarretaria uma drástica diminuição da simplicidade da linguagem e criaria "pontos obscuros" ao programador e, por isso, deve ser desconsiderada.

Uma avaliação mais criteriosa deste problema evidencia o fato de que ele é causado pela tentativa de utilização de uma única "abstração externa" em várias abstrações co-relacionadas de $\mathcal{LL}$, induzindo ao raciocínio de que o mecanismo de herança talvez não seja o mais adequado para reutilização de código em $\mathcal{LL}$. A solução, neste caso, seria apenas permitir correspondência biunívocas entre abstrações $\mathcal{LL}$ e abstrações externas.

A herança poderia, então, ser substituída por uma mecanismo que permitisse a inclusão de métodos a tipos já existentes (*extensão de tipos*). Sem sombra de dúvida isto pode acarretar conflitos, uma vez que um mesmo tipo pode ser estendido de várias formas distintas em vários módulos, mas esta é uma idéia que talvez traga bons frutos se analisada em atualizações futuras da linguagem.

2.4 Declaração de procedimentos e funções

A declaração de rotinas é constituída por um *cabeçalho*, que especifica como são invocadas, e por um *corpo*, que as implementam (figura 2.7)

```
procedure Ident (arg1; arg2; ...) is
   (* corpo *)
endprocedure

function Ident (arg1; arg2; ...) Restrição is
   (* corpo *)
endfunction
```

Figura 2.7: declaração de rotinas.

O cabeçalho de rotinas é semelhante ao de métodos. Seus parâmetros formais (se existentes) e valor de retorno (no caso de funções) podem apresentar restrições de tipo (verificadas em tempo de execução[4]).

[4]Um otimizador de código poderá eliminar algumas verificações redundantes.

Nomes de rotinas podem ser sobrecarregados. Apenas são consideradas conflitantes duas (ou mais) declarações de rotinas de mesmo nome, mesmo papel sintático (procedimento ou função) e mesma aridade. Porém, não constitui conflito a existência de declarações de métodos procedimentais (métodos funcionais) de mesmo nome e mesma aridade que procedimentos (funções). Em caso de invocações, os últimos precedem os primeiros.

O corpo de uma rotina, assim como o de um método, é implementado por uma *cadeia de implementação* ou por comandos $\mathcal{LL}$ precedidos, opcionalmente, por declarações de variáveis.

Cadeias de implementação, usadas apenas em módulos de ligação,[5] são seqüências de comandos da linguagem-hospedeira que definem completamente o corpo de uma sub-rotina, isto é, um corpo, ou é composto por comandos $\mathcal{LL}$ (incluindo declaração de variáveis), ou é constituído de uma única cadeia de implementação (figura 2.8). É este mecanismo que permite a utilização de bibliotecas externas em programas $\mathcal{LL}$.

```
linking module M is

    procedure P (X;Y) is
        'implP();'
    endprocedure;
    ...
endmodule
```

Figura 2.8: cadeia de implementação.

Estas duas restrições têm por objetivo evitar que o programador se utilize dos efeitos de comandos $\mathcal{LL}$ sobre o sistema de execução (particularmente sobre a pilha de execução) para implementar seus algoritmos, com prejuízo para a legibilidade e tornando o programa dependente da implementação do compilador, e para advertir ao compilador que possíveis otimizações de código podem ser incorretas. Dois exemplos ilustram cada um destes problemas.

Na tradução de um comando **case** (figura 2.9) a expressão *expr* é avaliada e o objeto resultante 0_r é colocado no topo da pilha de execução. Para que se verifique se a i-ésima opção é aplicável, o objeto 0_r é duplicado[6] (0_d) e comparado com o resultado da avaliação de $expr_i$ (0_i). Em caso afirmativo os comandos

[5] *linking modules*

[6] Na realidade apenas o descritor do objeto é duplicado.

da i-ésima opção são executados. Neste momento, 0_d e 0_i foram consumidos, restando 0_r no topo da pilha. Um programador, sabendo disto, poderia querer fazer uso deste valor. Se, por algum motivo, o algoritmo de tradução do comando **case** fosse alterado e 0_r fosse armazenado em algum outro lugar fora da pilha, o programa perderia sua correção.

case *expr* **of**
  |$expr_1$ ⇒ ...
  |$expr_2$ ⇒ ...
  ...
endcase

Figura 2.9: comando **case**.

Impedindo-se, portanto, que código *LL* se misture com cadeias de implementação em corpos de sub-rotinas, espera-se que o programador de interface[7] se utilize do sistema de execução da linguagem apenas para acessar parâmetros e retornar valores.

O segundo problema diz respeito a possíveis otimizações de código. Seja, por exemplo, o tipo *integer*. Suponha que um possível otimizador conheça sua representação interna e que faça uso desta informação. Considere, ainda, o tipo *Int*, sub-tipo de *integer*. Uma vez que não existem mecanismos para acréscimos de campos de dados na herança de *LL*, o otimizador poderia deduzir que a representação interna dos dois tipos seja a mesma. Isto pode não ser verdade caso algum construtor de *Int* crie um objeto de estrutura distinta de um objeto do tipo *integer*. Assim, distingüir módulos de ligação de módulos ordinários tem por objetivo advertir ao compilador que ele não possui completo domínio do sistema de execução.

2.5 Variáveis, constantes e parâmetros formais

Variáveis são nomes dados, através do comando de atribuição, a objetos resultantes de avaliação de expressões.[8] São locais ao escopo onde são utilizadas e podem denotar objetos de quaisquer tipos. Opcionalmente podem ser declaradas (figura 2.10).

[7] Ligação entre *LL* e bibliotecas externas.

[8] Passagem de parâmetros e o comando **forall** são definidos a partir do comando de atribuição.

var $v_1, v_2, \ldots : T$
var $v_1, v_2, \ldots <: T$
var $v_1, v_2, \ldots$

Figura 2.10: declaração de variáveis.

Restrições de tipo aplicadas a variáveis indicam ao compilador que deve ser verificado (em tempo de execução) se o objeto satisfaz a restrição imposta à variável a que é atribuído. Esta informação pode ser usada por um possível otimizador de código. Caso uma declaração não especifique qualquer restrição, ou se a variável não for declarada, implicitamente é aplicada a restrição <: *root* (sempre verdadeira).

Constantes são variáveis conhecidas globalmente no módulo onde são declaradas ou importadas, porém só podem ser atribuídas uma única vez, quando de sua inicialização (figura 2.11).

const *C1* : *T* **is** *expr*
const *C2* <: *T* **is** *expr*
const *C3* **is** *expr*

Figura 2.11: declaração de constantes.

Restrições de tipo podem ser facultativamente aplicadas ao resultado das expressões que as inicializam. Vale observar que o termo *constante* se aplica à ligação duradoura entre o nome e o objeto. O valor do objeto, entretanto, pode ser alterado em resposta à aplicação de seus métodos.

Deve ser ressaltada a inexistência de variáveis globais. Esta característica não constitui, entretanto, séria restrição, pois a manutenção do estado global de um sistema pode ser efetuada por meio de constantes. Alguns incovenientes podem advir desta particularidade de $\mathcal{LL}$, provocados pelo fato de que a ligação do nome ao objeto é determinada ao início da execução do programa e não pode ser alterada posteriormente. Outra ausência a ser notada é a impossibilidade em declarar-se enumerações, que também pode ser suprida por constantes como, por exemplo, a constante *true* do tipo *boolean* do módulo *System*.

Pressupõe-se, todavia, que, pelas características dos programas que a linguagem visa atender, variáveis globais e enumerações não sejam essenciais.

A instanciação de *parâmetros formais* é definida em termos do comando de

atribuição, ou seja, o resultado da avaliação de um argumento é atribuído a seu respectivo parâmetro formal. Não são, entretanto, permitidas atribuições a parâmetros formais dentro de seu escopo, sob pena de diminuição da legibilidade do código.

2.6 Expressões

Expressões definem computações que produzem *objetos*. São construídas pela aplicação de operações a expressões mais simples. Operações podem ser invocações de funções (incluindo construtores) ou de métodos funcionais (incluindo operadores). Expressões simples são denotações, variáveis, constantes ou parâmetros formais.

Invocações a funções da forma:

$$qualified\_id(expr,expr,...)$$

podem denotar funções ou métodos-funcionais. No primeiro caso, a seleção da rotina a ser executada é feita estaticamente. No segundo, é selecionado dinamicamente o método do tipo do primeiro argumento e de mesma aridade da invocação. A inexistência deste método provoca erro de execução do programa. Se existir a possibilidade da invocação ser respondida por um método ou por uma função, a última tem precedência. Se um possível otimizador determinasse em tempo de compilação o tipo do primeiro argumento da invocação a um método [dC], a seleção poderia ser estática.

O nome *qualified_id* pode ser qualificado como:

$$tipo.nome$$
$$modulo.tipo.nome$$
$$modulo.nome$$

As duas primeira qualificações explicitam o tipo onde o método é definido. Neste caso a seleção é estática. As duas últimas indicam o módulo onde o tipo ou a função foram declarados.

Operadores são considerados métodos, portanto, seus significados dependem do tipo de seu primeiro argumento. Podem ser pré-fixos, infixos ou pós-fixos. Para permitir invocações qualificadas, operadores possuem nomes funcionais. As invocações abaixo são equivalentes.

$$x - y \equiv OperatorMinus(x,y)$$
$$x = y \equiv OperatorEqual(x,y)$$
$$- x \equiv OperatorMinus(x)$$
$$x! \equiv OperatorExclamation(x)$$

2.7 Comandos

Comandos $\mathcal{LL}$ seguem a tradição de linguagens como Pascal [JW88] e Modula [Wir82, Nel91], alguns deles, entretanto apresentam particularidades.

2.7.1 Atribuição

O comando de atribuição, da forma

$$x \leftarrow expr$$

significa que o objeto resultante ($\mathtt{0_r}$) da avaliação da expressão *expr* é associado à variável x, local ao escopo da atribuição. Particularmente em:

$$y \leftarrow x$$

y denota o mesmo objeto de x, isto é, x e y *compartilham* o objeto $\mathtt{0_r}$.

Esta abordagem tem por objetivo garantir que o custo da atribuição seja constante. Note que objetos de aplicações típicas de $\mathcal{LL}$ podem ser estruturados em vários níveis (listas cujos elementos são outras listas,...), implicando cópias recursivas. Caso se queira que objetos sejam efetivamente copiados devem ser providos métodos para tal fim, possibilitando, inclusive, a utilização da atribuição com cópia ($\Leftarrow$, seção 2.8).

A *minimalidade* foi um dos princípios básicos do projeto de $\mathcal{LL}$. Como conseqüência, a atribuição múltipla da forma:

$$x_1,x_2,\ldots,x_n \leftarrow f(\ldots) \quad (*\ f\ retorna\ n\ valores\ *)$$
$$y_1,y_2,\ldots,y_k \leftarrow expr_1,expr_2,\ldots,expr_k$$

não foi incluída. Porém, devido a sua elegância, talvez pudesse ser integrada em futuras atualizações da linguagem.

2.7.2 Comando condicional

O comando condicional tem o seguinte aspecto:

```
if expr1
   then comandos1
   elsif expr2 then comandos2
   elsif expr3 then comandos3
   ...
   else comandos
endif
```

As expressões $expr_1$,$expr_2$,... são avaliadas até que $true = expr_i$ produza a constante *true* definida no módulo *System*, quando, então, $comandos_i$ serão executados.

As partes **elsif** e **else** são opcionais. Caso nenhuma das expressões seja válida e a parte **else** seja inexistente, o comando **if** não terá qualquer efeito (exceto se ocorrerem efeitos laterais durante a avaliação das expressões).

Vale observar que o resultado da avaliação das expressões deve ser idêntico à constante *true* definida no módulo *System*, pois $\mathcal{LC}$ não conhece a representação do valor lógico verdadeiro.

2.7.3 Comando de seleção

O comando de seleção de $\mathcal{LC}$, embora semelhante, é mais geral que os encontrados em linguagens de programação tradicionais. Sua forma é:

case *expr* **of**
  |$expr_1$ ⇒ $comandos_1$
  |$expr_2$ ⇒ $comandos_2$
  ...
  else *comandos*
endcase

Seu significado pode ser razoavelmente descrito por um comando condicional. A variável X é um novo identificador não utilizado neste escopo.

$X \leftarrow expr_i$
if $X = expr_1$
  then $comandos_1$
  elsif $X = expr_2$ **then** $comandos_2$
  ...
  else *comandos*
endif

Porém, a avaliação das condições não é feita necessariamente nesta ordem, permitindo que um possível otimizador adote um algoritmo de tradução mais eficiente. Por conseqüência, se duas ou mais opções forem aplicáveis, não se pode predizer qual seqüência de comandos será efetivamente executada. O possível custo extra das comparações inerente ao mecanismo é justificado pelo aumento de seu poder de expressão, indo de encontro à filosofia de projeto da linguagem.

2.7.4 Comandos iterativos

$\mathcal{LL}$ apresenta um rico conjunto de comandos de iteração:

```
loop
   comandos
endloop

while expr do
   comandos
endwhile


repeat
   comandos
until expr

forall v in expr do
   comandos
endforall
```

O comando **loop** expressa a contínua repetição da seqüência de comandos, salvo se interrompida por um comando **exit** ou **return**.

Os comandos **while**, **repeat** e **forall** podem ser descritos em termos do comando **loop**:

```
loop  (* while *)
   if expr
      then comandos
      else exit
   endif
endloop

loop  (* repeat *)
   comandos
   if expr
      then exit
   endif
endloop

X ← expr;
loop  (* forall *)
   if Empty(X)
      then exit
      else v ← Next(X)
   endif
   comandos
endloop
```

O comando **forall** pressupõe que *expr*, quando avaliada, produza um objeto (atribuído à variável fictícia *X*) que responda aos métodos *Empty* e *Next*. Tais objetos são denominados *iteradores*, porém não possuem qualquer significado especial para *LL*. Convém ressaltar que este mecanismo contribui de forma acentuada para a expressividade da linguagem. Um tipo pode definir vários métodos que produzam iteradores distintos. Grafos, por exemplo, podem possuir iteradores que os percorram em profundidade ou largura. Árvores podem definir iteradores que as percorram em pré-ordem, in-ordem ou pós-ordem.

2.7.5 Comandos de interrupção

Dois comandos alteram o fluxo normal de um programa *LL*. O comando

return *expr*

interrompe a execução de uma sub-rotina, retornando ao invocador o objeto resultante da avaliação de *expr*. Se a sub-rotina for um procedimento ou um método procedimental, *expr* deve ser omitida. O comando

exit

por sua vez, encerra a execução do comando iterativo mais interno. Não é permitida sua utilização em outro contexto.

2.7.6 Invocação de procedimento

Uma invocação, da forma

qualified_id(expr,expr,...)

pode denotar um procedimento ou um método procedimental. No primeiro caso, a seleção da rotina a ser executada é feita estaticamente. No segundo, é selecionado dinamicamente o método do tipo do primeiro argumento e de mesma aridade da invocação. A inexistência deste método provoca erro de execução do programa. Se existir a possibilidade da invocação ser respondida por um método ou por um procedimento, o último tem precedência. Se um possível otimizador determinasse em tempo de compilação o tipo do primeiro argumento da invocação a um método [dC], seleção poderia ser estática.

O nome *qualified_id* pode ser qualificado, como:

tipo.nome
modulo.tipo.nome
modulo.nome

As duas primeira qualificações explicitam o tipo onde o método é definido. Neste caso a seleção é estática. As duas últimas indicam o módulo onde o tipo ou o procedimento foram declarados.

2.8 Extensões sintáticas

A fim de aumentar a legibilidade de um programa, $\mathcal{LL}$ define construções sintáticas (*sintaxe estendida*) que são interpretados como invocações a pseudofunções (seção 2.9.1) ou a métodos.

Expressões $\{expr_0, expr_1, \ldots\}$, $<<expr_0, expr_1, \ldots>>$ ou $[[expr_0, expr_1, \ldots]]$, usadas para construção de objetos estruturados, são, respectivamente, interpretadas por invocações adequadas às pseudofunções *BraceDenotation*, *AngleBracketDenotation* ou *BracketDenotation*.

Componentes de objetos estruturados podem ser selecionados utilizando-se construções sintáticas tradicionais, desde que seus tipos definam os métodos adequados. A equivalência entre as notações sintáticas alternativas e seus respectivos métodos é definida pela tabela abaixo:

$expr_0[expr_1, expr_2, \ldots]$	$Value(expr_0, expr_1, expr_2, \ldots)$
$expr\$ident$	$ident(expr)$
$expr_1\$ident \leftarrow expr_2$	$Update\_ident(expr_1, expr_2)$
$expr_0[expr_1, expr_2, \ldots] \leftarrow expr$	$Update(expr_0, expr_1, expr_2, \ldots, expr)$

Como descrito na seção 2.7.1, um comando de atribuição pode provocar o *compartilhamento* de objetos. Uma atribuição da forma:

$$X \Leftarrow expr$$

é interpretado como uma invocação ao método *Copy*:

$$X \leftarrow Copy(expr)$$

Vale enfatizar que extensões sintáticas tratadas como invocações a métodos podem ser utilizadas com objetos de tipos que definam os métodos adequados, não importando seus significados.

2.9 Módulo *System*

$\mathcal{LL}$ foi idealizada a fim de prover uma estrutura sintática adequada para que projetistas de algoritmos utilizassem de forma fácil, limpa e rápida bibliotecas (inclusive as escritas em outras linguagens) na implementação de seus algoritmos.

$\mathcal{LL}$ deve ser flexível suficiente para suprir as necessidades específicas de cada comunidade de usuários no que se refere à representação de valores típicos de suas aplicações. Por exemplo, uma comunidade que trabalhe com aritmética exata pode desejar que as denotações inteiras e reais possuam representações internas distintas das comumentes utilizadas por outras comunidades; pesquisadores em processamento de textos gostariam que a representação de cadeias de caracteres contivesse particularidades não encontradas em geral. Por esse motivo $\mathcal{LL}$ não interpreta denotações, apenas as reconhece e, em tempo de execução, invoca rotinas providas pelo usuário (*pseudofunções*) que criam suas representações internas. Por conseqüência, não são predefinidos quaisquer tipos.

Por outro lado, algumas construções sintáticas se utilizam de valores de alguns tipos. São elas:

- A avaliação de expressões condicionais faz uso da implementação do valor lógico verdadeiro (constante *true*) para poder discernir entre a veracidade ou falsidade de uma condição estabelecida. O valor verdadeiro deve ser instância de algum tipo: *boolean*, por convenção;
- a hierarquia de tipos em $\mathcal{LL}$ pressupõe um tipo raiz (*root*);
- variáveis, quando não explicitamente inicializadas, possuem um valor específico que reflita tal fato. Este valor é identificado pela constante de nome *UndefinedValue* do tipo *undefined*.

As declarações de tipos, constantes e pseudofunções necessárias para que programas $\mathcal{LL}$ sejam operacionais devem estar descritas no módulo *System*, importado automaticamente por qualquer outro módulo através da cláusula **from** *System* **import all**, salvo determinação contrária. Este módulo pode conter outras declarações que tornem a tarefa de programação mais confortável.

É importante enfatizar que o módulo *System* pode ser alterado, ou mesmo substituído, a fim de se adequar à comunidade de usuários que farão uso da linguagem e às bibliotecas que utilizam. Uma experiência neste sentido, inclusive, já foi desenvolvida. Um módulo *System* alternativo foi implementado baseado na biblioteca **Leda**. Sua descrição pode ser encontrada em [PB].

2.9.1 Pseudofunções

Pseudofunções é, como já dito, o mecanismo pelo qual denotações são interpretadas, ou seja, podem ser enxergadas como construtores de tipos básicos.[9]

Suas declarações[10] indicam suas implementações. Considere as declarações a seguir:

```
(*$ BEGIN_PSEUDOFUNCTIONS *)

    IntegerDenotation(str): integer  == implSystemIntegerDenotation
    ProcRef(proc,int): procref       == implSystemProcRef
    FuncRef(func,int): funcref       == implSystemFuncRef

(*$ END_PSEUDOFUNCTIONS *)
```

A primeira pseudofunção é utilizada para interpretação de uma denotação inteira. Seu significado é: a rotina *implSystemIntegerDenotation*, escrita na linguagem hospedeira, receberá uma cadeia de caracteres (da linguagem hospedeira) e retornará um objeto $\mathcal{LL}$ do tipo *integer*. As especificações *int*, *str*, *proc* e *func* são estruturas de dados conhecidas pelo compilador e pelo programador de interfaces. A especificação *int* denota uma valor inteiro, *proc* e *func* denotam referências a rotinas.

Pseudofunções que se referem a denotações e a pseudofunção *TypeTag* são utilizadas internamente pelo compilador, não podendo ser invocadas ou exportadas num programa $\mathcal{LL}$. Outras pseudofunções (*ProcRef*, por exemplo) são tratadas de modo similar a funções $\mathcal{LL}$ (é permitida invocação, exportação e importação), porém a avaliação de seus argumentos e suas conseqüentes atribuições aos parâmetros formais é feita de forma distinta. Não é utilizado o sistema de execução de $\mathcal{LL}$, mas sim, o da linguagem hospedeira.

2.9.2 Exemplo de um módulo System típico

A seguir é apresentado o esquema de um módulo *System* padrão e de algumas implementações externas.

[9] Tipos básicos em outras linguagens seriam predefinidos.

[10] Delimitadas pelos pragmas *BEGIN_PSEUDOFUNCTIONS* e *END_PSEUDOFUNCTION*.

Módulo System

linking module *System* **is**

export
false, true, UndefinedValue, ProgramParameters, SystemError, root, undefined, integer, real, boolean, string, range, array, tuple, procref, funcref, typetag, ProcRef, FuncRef;

```
(*$ BEGIN_PSEUDOFUNCTIONS *)
   IntegerDenotation(str): integer        == implIntegerDenotation
   RealDenotation(str): real              == implRealDenotation
   StringDenotation(str): string          == implStringDenotation
   BraceDenotation(int): array            == implBraceDenotation
   BracketDenotation(int): array          == implBracketDenotation
   AngleBracketDenotation(int): tuple     == implAngleBracketDenotation
   TypeTag(int,int): typetag              == implTypeTag
   ProcRef(proc,int): procref             == implProcRef
   FuncRef(func,int): funcref             == implFuncRef
   ProcRef(proc): procref                 == implProcRef
   FuncRef(func): funcref                 == implFuncRef
(*$ END_PSEUDOFUNCTIONS *)

(* Constantes *)
   const false: boolean is boolean.False();
   const true: boolean is boolean.True();
   const UndefinedValue: undefined is undefined.New();

(* Funções *)
   function ProgramParameters(): array is
      'implProgramParameters();'
   endfunction;

(* Procedimentos *)
   procedure SystemError(s <: string) is
      'implSystemError();'
   endprocedure;
```

```
(* Tipos - veja especificação do tipo integer a seguir *)

    type root is ...;
    type undefined is ...;
    type integer is ...;
    type real is ...;
    type boolean is ...;
    type string is ...;
    type range is ...;
    type array is ...;
    type arrayValues is ...;
    type arrayItems is ...;
    type tuple is ...;
    type tupleValues is ...;
    type procref is ...;
    type funcref is ...;
    type typetag is ...;
endmodule
```

Tipo integer

type *integer* **is**

constructors

|*Max()*	⇒ *'implIntegerMax();'*
|*Min()*	⇒ *'implIntegerMin();'*

functions

|*ToReal(x <: integer): real*	⇒ *'implIntegerToReal();'*
|*ToString(x <: integer): string*	⇒ *'implIntegerToString();'*
|*Pred(x <: integer) <: integer*	⇒ *'implIntegerPred();'*
|*Succ(x <: integer) <: integer*	⇒ *'implIntegerSucc();'*

operators

|*(− x <: integer) <: integer*	⇒ *'implIntegerUnaryMinus();'*
|*(+ x <: integer) <: integer*	⇒ *'implIntegerUnaryPlus();'*
|*(x <: integer + y <: integer) <: integer*	⇒ *'implIntegerPlus();'*
|*(x <: integer − y <: integer) <: integer*	⇒ *'implIntegerMinus();'*
|*(x <: integer * y <: integer) <: integer*	⇒ *'implIntegerTimes();'*
|*(x <: integer* **div** *y <: integer) <: integer*	⇒ *'implIntegerDiv();'*
|*(x <: integer* **mod** *y <: integer) <: integer*	⇒ *'implIntegerMod();'*
|*x <: integer / y) <: real*	⇒ *'implIntegerDivide();'*
|*(x <: integer ↑ y <: integer) <: integer*	⇒ *'implIntegerExp();'*
|*(x <: integer ≠ y <: boolean) <: boolean*	⇒ *'implIntegerNotEqual();'*
|*(x <: integer < y <: integer) <: boolean*	⇒ *'implIntegerLess();'*
|*(x <: integer <= y <: integer) <: boolean*	⇒ *'implIntegerLeq();'*
|*(x <: integer > y <: integer) <: boolean*	⇒ *'implIntegerGreater();'*
|*(x <: integer >= y <: integer) <: boolean*	⇒ *'implIntegerGeq();'*
|*(x <: integer .. y <: integer) <: integer*	⇒ **return** *range.New(x,y);*

endtype

Implementações externas

Nos trechos de código abaixo rotinas prefixadas por `LLlib` constituem a parte do sistema de execução manipulável por implementadores de interfaces. `LLlib_push`, por exemplo, é equivalente a rotina `RTS_push` do sistema de execução (seção 4.4.1); `LLlib_pop` é equivalente a `RTS_pop` e assim sucessivamente.

Métodos e tipos são referenciados através de códigos dentro do sistema de execução de $\mathcal{LL}$, armazenados em variáveis globais. O programador de interfaces, por vezes, necessitará ter acesso a essas variáveis cujos nomes codificam informações como nome da método ou tipo, aridade, módulo. `_TYPE_CODE`, portanto, é uma macro que facilita a correta codificação destes nomes.

```
void implSystemIntegerDenotation(str s)
{
   LLlib_push(LLlib_create_descriptor(_TYPE_CODE(System,integer),
              (void *)atoi(s) ));
}

void implIntegerMax()
{
   LLlib_push(LLlib_create_descriptor(_TYPE_CODE(System,root),
            (void*)MAXINT));
}

void implIntegerPlus()
{
   int    x,y;

   if (!LLlib_subtypeof(RTS_get_top_minus_code(0),
             _t_System_integer_c))
   {
      cerr << "Conflito de tipos (IntegerPlus).  "
           << Programa abortado"
           << endl;
      exit(1);
   }

   if (!LLlib_subtypeof(RTS_get_top_minus_code(1),
       _t_System_integer_c))
   {
      cerr << "Conflito de tipos (IntegerPlus).  "
           "Programa abortado"
           << endl;
      exit(1);
   }

   y = (int) LLlib_get_pointer(LLlib_pop());
   x = (int) LLlib_get_top_pointer();

   LLlib_set_top_pointer((void*) x+y);
}
```

Figura 2.12: implementações externas.

Capítulo 3

Comparações com outras Linguagens

O principal objetivo de $\mathcal{LL}$ é prover um ambiente adequado para implementação eficaz de algoritmos bastante especializados que façam uso extensivo de bibliotecas de algoritmos e de estruturas de dados. Eficácia, neste contexto, significa implementação fácil, rápida e pouco sujeita a erros.

Acredita-se que este objetivo possa ser atingido pela conjugação dos princípios de projeto de $\mathcal{LL}$:

- *simplicidade* e *clareza*: a linguagem deve possuir uma quantidade reduzida de conceitos que devem ser dominados pelo programador. Estes conceitos, por sua vez, devem ser simples e claros. Minimalidade, simplicidade e clareza tendem a diminuir a possibilidade de inserção de erros durante a programação devido a complexidade excessiva da linguagem.

- *facilidade de utilização de bibliotecas*: uma linguagem de programação tem pouca utilidade se não provê bibliotecas de programas e estruturas de dados abrangentes. Por outro lado, $\mathcal{LL}$ não foi concebida para exercitar novos conceitos, outrossim, utiliza conceitos difundidos no âmbito de linguagens de programação descendentes de Pascal. Assim, não é justificável a completa reprogramação de bibliotecas já tradicionais. $\mathcal{LL}$, portanto, provê mecanismos que permitem a utilização de bibliotecas escritas em outras linguagens.

Uma vez que o propósito de $\mathcal{LL}$ é consolidação, em detrimento a qualquer tendência revolucionária, foram emprestadas de outras linguagens de programação

alguns conceitos e mecanismos. Faz-se, então necessário relacioná-los e avaliar em que medida vão de encontro à filosofia de $\mathcal{LL}$.

Três tópicos são abordados: controle de visibilidade entre entidades do programa, a maneira pela qual os dados são estruturados e a forma como são manipulados.

3.1 Controle de Visibilidade

Mecanismos que possibilitem uma divisão lógica do programa de forma coerente (isto é, definir agrupamentos de entidades relacionadas) e que permitam apenas acessos consistentes a estas entidades (controle de visibilidade) tendem a tornar o programa mais legível e menos propenso a erros de implementação.

O controle de visibilidade de $\mathcal{LL}$ é feito em três níveis: módulos, sub-rotinas e implementação.

Programas $\mathcal{LL}$ são compostos por módulos que provêem serviços ou que se utilizam de serviços de outros módulos. O modelo utilizado por $\mathcal{LL}$, similar ao de `Modula-2` [Wir82] e `Modula-3` [Nel91], se presta à organização coerente de bibliotecas, agregando tipos, constantes e rotinas relacionados à prestação de um serviço específico, e ao controle da visibilidade destas entidades. Programar constitui, portanto, escolher módulos (provedores de serviços) que possam auxiliar na resolução do problema proposto e combinar seus serviços a fim de que o problema seja efetivamente solucionado.

Algumas linguagens de programação, como `Smalltalk` [GR83] e `CLU` [L+79], investem no conceito de *ambiente da linguagem*, isto é, qualquer nova entidade declarada passa a ser conhecida por todos os programas. O processo de programação de `Smalltalk`, por exemplo, consiste em especializar um tipo (classe) definindo um novo sub-tipo ao qual são possivelmente acrescentados novos dados e novas operações. Todas as abstrações de `CLU` (procedimentos, iteradores e clusters) são incrementalmente introduzidas em sua biblioteca (ambiente). Este ambiente possui um espaço de nomes com informações relativas às abstrações compiladas, possibilitando posterior utilização.

Esta abordagem não é interessante para programadores $\mathcal{LL}$, pois não contempla o processo de programação proposto. Dificulta, inclusive, a análise do comportamento de um programa perante alternativas de implementação de um módulo específico.

Tipos (classes) também são mecanismos de modularização e de restrição de acesso. Linguagens como `C++` [Str86], `Trellis` [S+85] e `C+@` [EF+91, Fle93] dispõem de mecanismos que restringem acesso a dados e a operações, separando

conceitualmente a especificação (interface) e a implementação das operações do tipo.

Tipos de $\mathcal{LL}$ são a principal forma de ligação de bibliotecas externas, portanto, de forma geral, levam ao extremo o conceito de *Tipos Abstratos de Dados*: a especificação é descrita em $\mathcal{LL}$ e a implementação é feita externamente em alguma linguagem hospedeira, sendo, neste caso, completamente invisível ao usuário do tipo.

Por fim, como na maioria das linguagens de programação, variáveis e parâmetros são apenas visíveis no interior das respectivas sub-rotinas.

3.2 Estruturação dos dados

Problemas na implementação de algoritmos podem ser causados pela má estruturação dos dados que utiliza ou por sua manipulação errônea. *Encapsulamento* é uma tentativa de evitar manipulações incorretas e de facilitar a manutenção do programa. Consiste em não permitir ao usuário de uma abstração de dados tirar proveito de sua representação interna.

Encapsulamento em $\mathcal{LL}$ se dá por meio de tipos aos quais são associadas operações através das quais suas instâncias podem ser manipuladas. A representação interna de um tipo é ordinariamente inacessível aos seus usuários.

Além de tentar previnir manipulações errôneas de dados através de encapsulamento, $\mathcal{LL}$ deve prover facilidades para que novas funcionalidades sejam acrescentadas a tipos disponíveis aos usuários, visto que uma de suas principais características consiste em uma grande utilização de bibliotecas durante o processo de programação, visando intensa reutilização de código. É neste contexto que emerge o conceito de herança em $\mathcal{LL}$.

Herança múltipla não é conceitualmente simples e, por isso, existem controvérsias sobre sua real utilidade. Esta opção, portanto, foi desconsiderada no projeto da linguagem. Tipos em $\mathcal{LL}$ possuem, então, um único ancestral (supertipo) direto.

O mecanismo de herança (tanto simples, quanto múltipla) pode ferir o conceito de encapsulamento desejado por linguagens baseadas ou orientadas a objetos na medida que permite que atributos de um tipo possam ser acessados por seus sub-tipos.

$\mathcal{LL}$, por esconder completamente a representação interna de seus objetos, não permite que o encapsulamento seja violado pela herança.

É interessante notar que a herança de $\mathcal{LL}$ visa criar novos tipos com novas operações (funcionalidades). Wirth em [Wir88b] segue o caminho oposto. Des-

creve um esquema de herança mais restrito — a extensão linear de tipos — pela qual um novo tipo é criado através da herança de campos de um super-tipo e pelo acréscimo de novos campos de dados. São definidas regras claras e precisas de atribuição de objetos (valores) de tipos estendidos a variáveis de tipos ancestrais. Este mecanismo é utilizado de `Oberon` [Wir88a]. `Oberon-2` [MW92] estende este mecanismo permitindo o atrelamento de operações aos tipos.

Em [Bro90] é descrito o modelo não linear de extensão de tipos, baseado no trabalho de Wirth, que permite que um tipo seja extensão direta de vários tipos ancestrais.

3.3 Estruturas de Controle

As estruturas de controle de $\mathcal{LL}$ são, em geral, bastante conhecidas. Algumas delas determinam as principais características da linguagem.

Entre elas talvez a que exerça mais influência é a atribuição e sua relação com o conceito de variáveis.

Variáveis são locais e não possuem tipo estático, apenas dão nomes a objetos. Como primeira conseqüência, os tipos de expressões apenas podem ser determinados em tempo de execução e, portanto, a seleção de métodos é feita dinamicamente. Isto envolve um gasto extra de tempo na invocação a métodos. É sabido, ainda, que a verificação estática dos tipos das expressões de um programa contribuem para sua correção. A ausência desta verificação poderia, desta forma, aumentar (ou deixar de diminuir) erros de lógica do programa.

Nenhum dos dois fatos levantados é determinante para aplicações às quais $\mathcal{LL}$ se destina. Como a linguagem se baseia em forte utilização de bibliotecas, espera-se que o tempo gasto para a invocação de um método seja desprezível em relação a sua execução. Ao lado disso, $\mathcal{LL}$ não é uma linguagem para programação de grandes sistemas, mas sim, destina-se à implementação de algoritmos isolados, fruto do trabalho de um pesquisador, novamente, com forte utilização de bibliotecas bastante poderosas. Daí ser razoável supor que programas contenham poucas linhas de código. Tendo em vista este horizonte, dizer que incompatibilidades de tipos podem ser encontradas sem grandes dificuldades através da inspeção de código não parece descabido. Vale notar que a verificação da compatibilidade de tipos em uma expressão, embora não seja estática, não é esquecida, mas, ao contrário, é feita em tempo de execução. Assim a correção entre o relacionamento dos tipos de uma expressão ainda é assegurada.

Esta abordagem, também apresentada por Smalltalk, permite maior generalidade dos algoritmos e, portanto, maior reutilização de código.

Outra estrutura de controle de $\mathcal{LL}$ também importante, é o comando **forall** que faz aflorar o conceito de *iterador*.

Iteradores, encontrados em várias linguagens de programação, é um mecanismo de grande poder de expressão. Não são raros algoritmos que apresentem construções da forma:

para todos os elementos do conjunto faça ...
para todos os nós do grafo G faça ...
para todos as sub-divisões do plano P faça ...

Tais construções sub-entendem um conjunto diferente de elementos, possivelmente vazio, do qual é consultado um elemento a cada passo da iteração.

Linguagens como `Pascal` e `C` provêem iteradores apenas sobre valores escalares. O algoritmo pode, neste caso, ser descaracterizado se a estrutura a ser percorrida for complexa. Em outro extremo existem linguagens de programação, como `CLU` e `Trellis`, nas quais iteradores são suportados e são implementados como co-rotinas retomadas a cada passo da iteração.

$\mathcal{LL}$ possui uma formulação intermediária entre estes dois pólos. Em princípio, o conceito de iterador não é definido, isto é, não há um objeto de algum tipo especial exigido pelo comando **forall**, porém, a semântica deste comando incentiva a criação de objetos com funcionalidade de iteradores. Este mecanismo de iteração de $\mathcal{LL}$ provê uma flexibilidade que permite maior clareza e simplicidade dos programas, diminuindo o hiato semântico entre o algoritmo e sua implementação.

Um terceiro conceito que pretende incrementar clareza, simplicidade e reusabilidade de código de programas $\mathcal{LL}$ é o polimorfismo.

Rotinas polimórficas são aquelas cujos argumentos podem denotar objetos de vários tipos. A única especificidade de uma função que ordene um conjunto de dados, por exemplo, reside na forma de comparar dois objetos deste conjunto e determinar o maior entre eles. Exceto por este detalhe, a função funciona independente do tipo dos argumentos. Caso sua implementação não seja polimórfica, devem ser criadas tantas funções quanto forem os tipos dos objetos a serem ordenados. Tal abordagem apresenta reusabilidade nula e, obviamente, dificulta a manutenção do programa. A simplicidade fica comprometida uma vez que a simplicidade conceitual do algoritmo não é refletida no programa, isto é, o algoritmo exige que um conjunto de dados seja ordenado, mas o programador deve escolher (ou implementar) uma função específica para tipos de dados particulares.

Cardelli e Wegner [CW85] classificam e descrevem várias formas de polimorfismo. Em um primeiro nível as formas de polimorfismo são classificadas em *universal* e *ad hoc* segundo a homogeneidade das estruturas de dados que mani-

pulam. Posteriormente, cada uma destas categorias é sub-dividida pela maneira que o polimorfismo é obtido.

No polimorfismo *universal* a estrutura dos tipos dos dados manipulados é uniforme. Isto permite que funções polimórficas universais possam atuar sobre um conjunto quase ilimitado de tipos de objetos e que o código executado seja o mesmo, independente dos tipos dos argumentos. O polimorfismo universal pode ser dividido em *paramétrico* e *de inclusão*.

Uma *função polimórfica paramétrica*, também denominada *função genérica*, possui um parâmetro implícito ou explícito que determina o tipo do argumento de cada invocação.

O *polimorfismo de inclusão* permite que um objeto seja visto como pertencente a várias classes diferentes, não necessariamente disjuntas. Modela o mecanismo de herança de linguagens orientadas a objetos.

Quando a estrutura dos tipos dos argumentos da função polimórfica é heterogênea, manifesta-se o *polimorfismo ad hoc*. Nesta espécie de polimorfismo, as funções funcionam (ou parecem funcionar) para argumentos de vários tipos, porém, seu comportamento pode ser bastante diverso para cada um dos tipos dos argumentos.

O polimorfismo ad hoc é dito ser *de sobrecarga*, quando um mesmo nome é usado para denotar diferentes rotinas e o contexto é usado para decidir qual das rotinas é referida por uma invocação específica. Um passo de pré-processamento poderia eliminar toda sobrecarga do programa, atribuindo diferentes nomes para diferentes funções.[1]

Outra forma de polimorfismo ad hoc é a *coerção*, isto é, uma operação semântica que converte o argumento em um tipo aceitável pela função invocada.

Objetos $\mathcal{LL}$ possuem representação uniforme, permitindo que rotinas apresentem polimorfismo universal. A função de ordenação é um bom exemplo de polimorfismo paramétrico. O tipo implícito da função é o tipo dos objetos do conjunto a ser ordenado.

$\mathcal{LL}$ possui o conceito de sub-tipo e de herança e, portanto, apresenta polimorfismo de inclusão. Ainda, várias rotinas podem ser declaradas com o mesmo nome, verificando-se a sobrecarga.

É importante observar a impossibilidade de coerção co-existir com a linguagem. $\mathcal{LL}$, por não conhecer a representação interna de seus objetos, não é capaz de prover conversões automáticas entre tipos de objetos e, nem mesmo, provê mecanismos para que seus usuários implementem tais rotinas, que seriam invocadas

[1] Existem controvérsias sobre a adequação em considerar sobrecarga como polimorfismo. Muitos consideram-na apenas como um problema de identificação de nomes.

automaticamente.

É interessante ainda observar que *LL* permite que rotinas e métodos sejam implementados através de comandos de outra linguagem (linguagem hospedeira).

Esta característica, embora a distancie das linguagens de programação da família de Pascal, aproxima *LL* de linguagens preocupadas com reutilização de código, como C+@, e de *little languages* que dependem de uma linguagem de suporte (por exemplo,Lua [F+94]).

Ambas linguagens permitem referências a funções e variáveis declaradas e implementadas (e compiladas) em C. C+@, inclusive, provê conversões automáticas de tipos dos argumentos quando uma função implementada externamente é invocada.

LL também permite implementações externas, mas ao contrário destes exemplos, não interpreta suas cadeias de implementação. Não se deseja que este mecanismo se torne uma técnica de programação ordinária. *LL* vislumbra dois tipos de programadores. O primeiro, o *programador final*, é o público alvo da linguagem. Seu objetivo é implementar um algoritmo utilizando um farto conjunto de bibliotecas. O segundo, o *programador de interfaces*, possui o mesmo *status* do projetista do compilador. Cabe a ele adequar o compilador ao ambiente do usuário final, criando interfaces entre as bibliotecas disponíveis no ambiente que o usuário final estava acostumado a trabalhar e a linguagem *LL*. Apenas neste momento, portanto, é justificada a utilização de cadeias de implementação.

Lua e C+@ não fazem esta distinção. Implementações externas nestas linguagens são encaradas como técnica de programação para, por exemplo, reutilização de código (C+@). Note, entretanto, que a reusabilidade é comprometida, pois é restringido o espectro de bibliotecas que podem ser migradas para C+@.

Por fim, vale observar que linguagens como Trellis e CLU apresentam açúcares sintáticos semelhantes aos de *LL*. Sua utilização diminui a complexidade do compilador uma vez que reduz o número de mecanismos que devem ser efetivamente implementados. Além disso, permite que a linguagem seja facilmente atualizada caso seja indentificada a necessidade de novas construções sintáticas.

Vale observar que no caso de *LL* a existência dos açúcares sintáticos é, em última análise, determinada pela inexistência de tipos primitivos. Isto obriga que quaisquer operações sobre objetos sejam realizadas através da invocação de métodos. A fim de que a legibilidade do programa não seja prejudicada, certas notações podem substituir a invocação de métodos específicos.

Capítulo 4

Implementação do Compilador

Neste capítulo são abordados alguns aspectos sobre a implementação do compilador. Não é seu objetivo que se torne referência suficiente para manutenção do programa. Maiores detalhes podem ser encontrados em [BK94].

O primeiro protótipo do compilador $\mathcal{LL}$ é dividido em cinco módulos: os analisadores léxico, sintático e semântico, um otimizador de invocações, e um gerador de código. Estes módulos, por sua vez, interagem principalmente com duas estruturas de dados: a *árvore de programa* e a *tabela de símbolos.*

O *analisador léxico* foi codificado com o auxílio da ferramenta `flex` [Pax] e tem como principal característica não interpretar cadeia de caracteres relativa a uma denotação. Outrossim, esta cadeia é armazenada na árvore de programa e seu significado será determinado apenas em tempo de execução, através de pseudofunções adequadas (seção 2.9.1).

O *analisador sintático*, baseado no método *LR*, foi gerado automaticamente pela ferramenta `bison`[Sta].

A *análise semântica*, por sua vez, é responsável por manter a tabela de símbolos, construir a árvore de programa, resolver sobrecarga de nomes e fazer algumas verificações semânticas. Vale observar que a árvore de programa sofre algumas manipulações durante sua construção, assim como alguns nomes extras são colocados na tabela de símbolos. Estes detalhes são tratados nas próximas seções.

Antes que a geração de código tenha início, é interessante que se determine, sempre que possível, o tipo do primeiro argumento da invocação de um método de forma a permitir que a determinação da instância do método que será ativada seja feita estaticamente, isto é, em tempo de compilação. Esta tarefa é feita por um *otimizador de invocações*, assunto de outra dissertação de mestrado [dC].

Por fim, o último módulo do compilador *LL* é o *gerador de código*. Caracteriza-se por ser facilmente substituído por outras implementações, gerando código em outra linguagem hospedeira (atualmente C++).

4.1 Estruturas de Dados

4.1.1 Tabela de Símbolos

A *tabela de símbolos* (TS) é composta de três estruturas: *vetor de nomes*,[1] onde cada nome é armazenado sem repetições; *descritor de nome*, com informações associadas a um nome específico, e uma *tabela de espalhamento*,[2] cujas entradas correspondem a listas de descritores. A relação entre estas estruturas é ilustrada pela figura 4.1, correspondendo às declarações:

const *X* **is** *<expr1>;*

procedure *P ( )* **is**
  X ← *<expr2>;*
endprocedure;

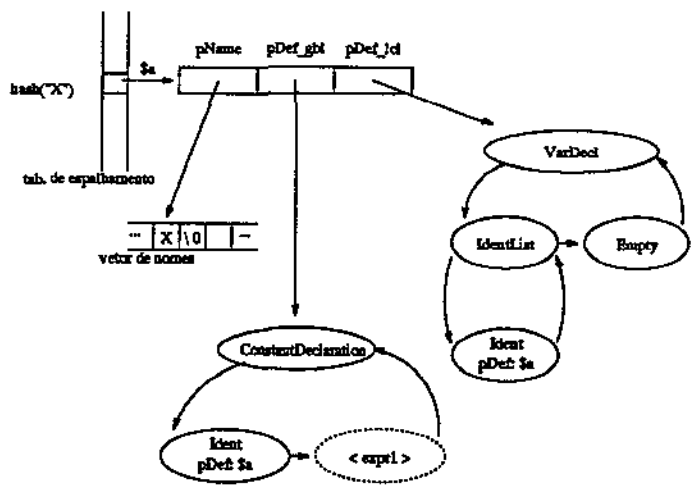

Figura 4.1: tabela de símbolos.

Internamente ao compilador, um nome é identificado por uma referência ao seu descritor. O processo de inserção de um nome consiste em verificar se o nome

[1] *String pool*
[2] *Hashing*

já existe; em caso afirmativo, é retornada uma referência a seu descritor; caso contrário, o nome é inserido no vetor de nomes, é criado um descritor que armazenará uma referência à localização do identificador dentro do vetor de nomes e, por fim, o descritor é inserido em alguma das listas da tabela de espalhamento, sendo retornada sua referência.

Embora sejam armazenados sem repetição na TS, é possível que alguns nomes (de procedimentos, por exemplo) sejam codificados de várias formas (a fim de que sobrecargas sejam diferenciadas) e inseridos na TS. Desta forma, a um nome específico podem corresponder vários descritores de nomes. Isto não contradiz a unicidade de armazenamento de identificadores, posto que cada descritor corresponde a uma codificação distinta do identificador.

Os descritores contêm, além da referência à localização do identificador no vetor de nomes, outras informações para a resolução de sobrecargas e para determinação de seu significado[3] dentro de um determinado escopo. Seus campos são suscintamente descritos a seguir:

- *pName:* referência à localização do identificador dentro do vetor de nomes;
- *pDef_lcl: binding* local do identificador;
- *pDef_gbl: binding* global do identificador;
- *P0, P1, F0, F1:* marcas para tratamento de sobrecarga;
- *pMod:* referência ao módulo onde está a declaração referente ao *binding* global do identificador.

4.1.2 Árvore de Programa

A *árvore de programa* (AP) é uma representação intermediária do programa em processo de compilação. É constituída de nós que armazenam informações sobre sua própria estrutura e a do programa.

Os principais campos de cada nó são:

- *Code:* código do nó; indica sua função sintática;
- *First:* referência ao primeiro filho;
- *Next:* referência ao próximo irmão; no caso de ser o último, aponta para o pai;

[3] *binding*

- *Last:* valor lógico verdadeiro se é o último nó em uma lista de irmãos;
- *pDef:* caso o nó corresponda a um identificador, faz referência a seu descritor na TS;
- *pChar:* se o nó refere a uma denotação, faz referência a respectiva cadeia de caracteres;
- *pOpt:* informações para otimização de invocação.

Além dessas, outras informações são armazenadas na AP, tais como: o número da linha de um comando (para facilitar a emissão de mensagens de erros de execução), número de argumento de uma invocação de rotina ou método, entre outras.

A construção é efetivada por, basicamente, três tipos de ações:

- criação de nó relativo a símbolos terminais, tais como: identificadores, denotações, etc. Note que símbolos terminais separadores (“,” , “;” , “:” , etc) não são inseridos na AP, pois não possuem qualquer significado semântico;
- construção de uma sub-árvore, representando alguma estrutura sintática da linguagem $\mathcal{LL}$ como, por exemplo, um comando condicional;
- construção de seqüências (listas) de estruturas sintáticas semelhantes como, por exemplo, lista de comandos, lista de identificadores, etc.

Os principais métodos de construção da AP são:

- `void Push_node (TreeNode *pT)`
- `void Build (NodeCode C, int N)`
- `void Create_list (NodeCode C, int Incr = 1)`
- `void Merge (NodeCode C, int Incr = 1)`

O cabeçalho `procedure Proc1 (X,Y) is` produzirá a sub-árvore da figura 4.2.

Inicialmente a AP reflete fielmente a estrutura da gramática da linguagem. Entretanto, o compilador $\mathcal{LL}$, durante a tradução de um programa, fará alterações nesta estrutura, visando, sobretudo, tornar as fases de análise semântica e geração de código mais homogêneas possíveis.[4] São elas:

[4]Isto é, sem muitos casos especiais a serem verficados.

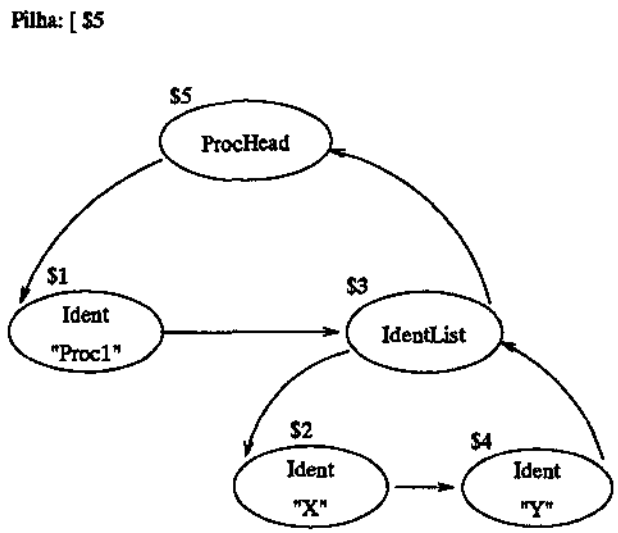

Figura 4.2: árvore de programa.

- A árvore de programa procura não representar a *sintaxe estendida* (seção 2.8). Por exemplo, uma atribuição de cópia ($A \Leftarrow expr$) é substituída pela invocação ao método *Copy* ($A \leftarrow Copy(expr)$). Isto permite que "açúcares sintáticos" sejam acrescidos a linguagem $\mathcal{LL}$ a custa de poucas alterações no compilador;

- o corpo de métodos e construtores são encapsulados por funções e procedimentos $\mathcal{LL}$;

- são criadas declarações de métodos funcionais equivalentes às declarações de operadores de um tipo, facilitando o tratamento de qualificação de nomes de operadores;

- para cada variável local não declarada dentro do escopo de uma rotina[5] ou método, é criada uma declaração implícita sem restrição de tipo. Isto é necessário para a determinação do *binding* local de nomes de variáveis (seção 4.2);

- se o módulo *System* não foi explicitamente importado, é inserida na AP uma sub-árvore relativa à declaração **from** *System* **import all** no módulo principal;

[5]Incluindo o corpo do módulo.

- a cada módulo importado corresponderá uma sub-árvore de programa contendo as declarações por ele exportadas.

4.2 Processo de Compilação

O processo de tradução de um módulo $\mathcal{LL}$ compreende basicamente quatro tarefas. A princípio é identificada a estrutura sintática do programa. Posteriormente são determinados os significados de seus nomes, são feitas algumas verificações sobre a consistência semântica do programa e, por fim, é realizada a tradução propriamente dita. Este processo segue os seguintes passos:

1. Construção da árvore de programa:

   (a) construção da árvore de programa sem definição de *bindings*, de forma que nós relativos a identificadores façam referência a sua respectiva entrada na tabela de símbolos (um nome corresponde uma única entrada na tabela de símbolos). Note que a *sintaxe estendida*, sempre que possível, é substituída pela *sintaxe estrita* na árvore de programa. Nesta fase, corpos de métodos e de construtores são convertidos em procedimentos ou funções, refletindo na árvore de programa. São geradas declarações de métodos funcionais para operadores;

   (b) inserção de entidades importadas (tipos, constantes, procedimentos, funções e módulos) na árvore de programa como se tivessem sido declaradas no módulo corrente, porém sem seus corpos, sejam eles expressões que inicializam as constantes ou comandos de rotinas ou métodos.

2. *Binding* e geração de código

   (a) *binding* global: são percorridas as declarações de constantes, procedimentos, funções, pseudofunções, módulos e tipos definindo o *binding* global desses nomes na TS. Neste passo, durante o percurso de declarações de procedimentos e funções, são tratadas a sobrecargas de nomes de rotinas;

   (b) verificações semânticas: a árvore de programa é percorrida a fim de garantir que o programa é semanticamente consistente. É assegurado que, por exemplo, métodos invocados qualificadamente tenham sido declarados, não exista circularidade na declaração de módulos e de

tipos, etc. Tais validações não poderiam ser executadas anteriormente, uma vez que apenas a partir do passo anterior nomes globais possuem seus significados determinados. Por outro lado, estas verificações não foram adiadas até a geração de código, economizando um percurso na árvore de programa, pois considera-se desejável isolar a geração de código de quaisquer outras preocupações, facilitando a substituição do módulo de geração por outros módulos que produzam código em outras linguagens hospedeiras (seção 4.3), aumentando a quantidade de bibliotecas utilizáveis dentro do ambiente $\mathcal{LC}$.[6]

(c) *binding* local e geração de código: para cada corpo de procedimento, função, especificação de método e para o corpo do módulo são executados os seguintes passos:

i. Procurar referências a nomes:
Se o nome está do lado esquerdo de uma atribuição, ou é uma variável de controle de um comando **forall** (iterador), ou, ainda, está sendo declarada como variável ou parâmetro formal, temos uma *declaração local* do nome. Nos dois primeiros casos, é criada uma declaração para esta variável (caso não exista), colocando-a na árvore de programa como se a declaração tivesse sido feita pelo programador (com restrição <: *root*) e fazendo o "binding" local deste nome se referir a esta declaração. Nos outros casos, apenas faz-se o *binding* local do nome se referir às declarações já existentes na árvore de programa.
Se, por outro lado, o nome é usado como nome de uma rotina ou como expressão, temos os seguintes casos:

- o nome é apenas declarado globalmente: utiliza-se o seu "binding" global;
- o nome é declarado tanto globalmente quanto localmente: o "binding" depende do contexto da utilização:
  - contexto é de invocação de rotina: utiliza-se o "binding" global;
  - contexto é de variável: utiliza-se o "binding" local.

ii. Gerar código para o corpo da sub-rotina na linguagem hospedeira;

iii. Percorrer as declarações de variáveis e parâmetros formais na árvore de programa, desfazendo os "bindings" locais.

[6] Pretende-se futuramente implementar um módulo gerador que produza código `Modula-3`.

4.3 Arquivos Intermediários

O compilador $\mathcal{LL}$ não produz diretamente código-objeto, pelo contrário, traduz o código $\mathcal{LL}$ para uma linguagem intermediária de alto nível (*linguagem hospedeira*) que é posteriormente traduzida por um compilador adequado (*compilador hospedeiro*).

Embora menos eficiente, esta abordagem justifica-se por duas razões básicas:

- a utilização de linguagens compatíveis, em termos de sistema de execução, na geração de código-hospedeiro e na codificação de bibliotecas externas, provavelmente diminuirá drasticamente problemas de integração de bibliotecas externas ao ambiente $\mathcal{LL}$. Supõe-se que esta redução será mais evidente caso os compiladores do código-hospedeiro e da biblioteca externa forem os mesmos.

- programas $\mathcal{LL}$ podem ganhar em portabilidade e em otimização de código se a linguagem hospedeira possuir compiladores-otimizadores disponíveis em várias plataformas.

O compilador $\mathcal{LL}$ atualmente produz código `C++`, porém, o módulo gerador pode ser substituído a fim de produzir código em outras linguagens hospedeiras, permitindo a integração de $\mathcal{LL}$ com um espectro maior de bibliotecas externas.

Durante o processo de tradução de um módulo `m.ll` são gerados alguns arquivos intermediários que visam permitir a compilação em separado do módulo m ou a criação do arquivo executável. São eles:

- `m.lls`: contém informações que possibilitam a atualização da tabela de símbolos quando este módulo for importado. É utilizado pelo compilador $\mathcal{LL}$ (`LLC`).

- `m.h` e `m.cxx`: tradução do módulo m. Serão traduzidos pelo compilador hospedeiro (`HC`).

- `RTS.cxx`: define a parte do sistema de execução específico ao módulo `m.ll`.

A compilação de um módulo m, que importa os módulos $m_1,\ldots,m_k$, é representado pela figura 4.3.

O processo completo de compilação pode ser orientado à compilação em separado ou a produção do arquivo executável. No primeiro caso é apenas gerado código-objeto correspondente ao módulo m.

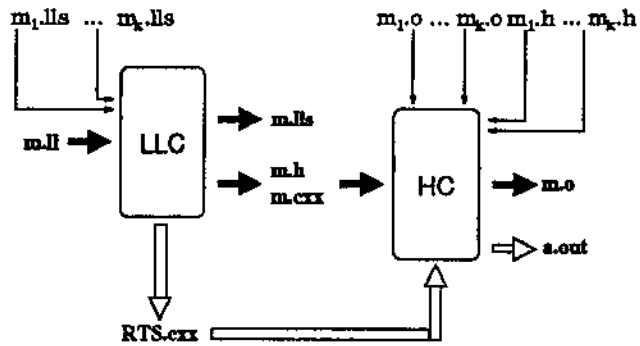

Figura 4.3: compilação de m.ll.

Entretanto, a produção do arquivo executável compreende várias inicializações, como definição de códigos de tipos e métodos (usado no processo de seleção dinâmica), incluídos no arquivo `RTS.cxx`, que apenas podem ser feitas nesta fase.

4.4 Sistema de Execução

O *Sistema de Execução* de *LL* é composto por duas partes distintas. A primeira, denominada *Sistema de Execução Básico*, é constituída por uma pilha de execução, por uma árvore de hierarquia de tipos e por rotinas que as manipulam. É comum a qualquer programa *LL*.

A segunda parte, o *Sistema de Execução Específico*, é responsável pela inicialização de variáveis de códigos de tipos e métodos, pelo mecanismo de seleção dinâmica e pela inicialização do programa. É específico para diferentes programas *LL*, visto que seus conjuntos de tipos e métodos são possivelmente distintos, o que implica códigos e mecanismos de seleção dinâmica particulares.

Internamente ao Sistema de Execução, métodos e tipos são identificados por códigos inteiros, porém, não podem ser enumerados antes da compilação final (geração do arquivo executável), uma vez que o conjunto dos tipos e métodos que integram o programa só pode ser conhecido neste momento. Por conseqüência, durante a compilação em separado de um módulo, tais códigos são referenciados através de variáveis globais, posteriormente inicializadas (na compilação final). Vale observar que o código de um método procedimental, por exemplo, deve ser o mesmo independente do tipo ou módulo onde tenha sido declarado.

4.4.1 Pilha de Execução

Um objeto $\mathcal{LL}$ (`LLtype`) é um registro com dois campos: uma marca com o tipo a que pertence, isto é, um código inteiro, e um apontador para representação interna do objeto. No caso de objetos "curtos" e imutáveis, como, por exemplo, os tipos *integer* e *boolean* do módulo *System*, o apontador pode ser substituído por seu próprio valor.

A *pilha de execução* armazena objetos $\mathcal{LL}$ e é usada na passagem de parâmetros, no retorno de resultados de funções e na avaliação de expressões. É manipulada pela utilização das operações abaixo:

- `void RTS_init_stack (int T)`: cria uma pilha de execução para, no máximo, T objetos;
- `int RTS_empty ()`: retorna "1" se pilha está vazia, "0", caso contrário;
- `void RTS_push (LLtype)`: empilha um objeto;
- `LLtype RTS_pop ()`: desempilha um objeto $\mathcal{LL}$;
- `void RTS_dup ()`: duplica o topo da pilha;
- `void RTS_dup_minus(int I)`: coloca no topo da pilha de execução uma cópia do I-ésimo elemento abaixo do topo. Observe que `RTS_dup_minus(0)` é equivalente a `RTS_dup()`;
- `LLtype RTS_top()`: retorna cópia do descritor do topo da pilha, sem desempilhá-lo;
- `LLtype RTS_top_minus (int I)`: retorna cópia do I-ésimo descritor abaixo do topo da pilha. Note que `RTS_top_minus(0)` tem o mesmo efeito de `RTS_top()`;
- `int RTS_get_top_code ()`: retorna código do descritor do topo da pilha;
- `void RTS_set_top_code (int C)`: atribui C ao codigo do descritor do topo da pilha;
- `int RTS_get_top_minus_code (int I)`: retorna código do I-ésimo descritor abaixo do topo da pilha. `RTS_get_top_minus_code(0)` é semelhante a `RTS_get_top_code()`;
- `void RTS_set_top_minus_code (int I,int C)`: atribui o código C ao I-ésimo descritor abaixo do topo da pilha;

- `int RTS_compare_identity (LLtype L1,LLtype L2)`: retorna 1 se os descritores L1 e L2 são idênticos, isto é, possuem o mesmo código e o mesmo apontador para representação interna;
- `void *RTS_get_top_pointer ()`: retorna o apontador da representação interna do descritor do topo da pilha;
- `void RTS_set_top_pointer (void *p)`: atribui p ao apontador da representação interna do descritor do topo da pilha;
- `void *RTS_get_top_minus_pointer (int I)`: retorna apontador da representação interna do I-ésimo descritor abaixo do topo da pilha;
- `void RTS_set_top_minus_pointer (int I,void *p)`: atribui p ao apontador da representação interna do I-ésimo descritor abaixo do topo da pilha;
- `int RTS_subtypeof (int C1, int C2)`: retorna "1" se o tipo de código `C1` é subtipo do tipo de código `C2` ou se `C1` e `C2` denotam o mesmo tipo; "0", caso contrário;
- `int RTS_strict_subtypeof (int C1, int C2)`: similar a `RTS_subtypeof`, porém, retorna "1" apenas se `C1` é estritamente subtipo de `C2`;
- `int RTS_same_type (int C1, int C2)`: retorna "1" se os tipos de código `C1` e `C2` denotam o mesmo tipo; "0", caso contrário;[7]
- `void RTS_run_time_error (char *Msg, char *Mod, int Line)`: imprime a mensagem `Msg`, indicando erro de execução na linha `Line` do módulo `Mod`, e interrompe a execução do programa;
- `void RTS_debug_run_time_error (char *Msg,char *Mod,int Line)`: exibe mensagem `Msg` indicando erro de execução no módulo `Mod` , na linha `Line`, além da cadeia de invocações;
- `int RTS_num_args ()`: retorna o número de argumentos passados para o programa;
- `char **RTS_args ()`: retorna um apontador para o primeiro argumento do programa.

[7] Ou seja, retorna "1" se `C1 == C2`.

A maioria destas operações pode ser utilizada pelo programador de interface, de forma a ser possível que implementações externas de rotinas e métodos tenham acesso aos argumentos de suas invocações, compreendam a estrutura interna de objetos $\mathcal{LL}$, criem novos objetos, enfim, interajam com o sistema de execução da linguagem. Note-se, portanto, que o programador de interfaces possui um grau de responsabilidade semelhante ao do projetista do compilador.

4.4.2 Seleção dinâmica

Seja a invocação a um método `m`$(e_1,\ldots,e_n)$. A implementação do método `m` que será executada dependerá do tipo do primeiro argumento e_1 e da aridade da invocação n. Como, via de regra, o tipo de e_1 só poderá ser determinado em tempo de execução, o *binding* de métodos deve ser dinâmico.[8]

A implementação do método `m`, neste caso, é procurada no tipo `T` do primeiro argumento. Se não for encontrada é procurada em todos super-tipos de `T`. Não sendo definido nem mesmo no tipo *root* é gerado um erro de execução.

A implementação da rotina de seleção dinâmica se dá em dois níveis: uma rotina de seleção (`BINDER`) implementada no sistema de execução específico, e várias rotinas de seleção, uma para cada tipo T (`_M_T_binder`) de um módulo M.

O procedimento `BINDER` (figura 4.4) determina o tipo do primeiro argumento da invocação do método e ativa a rotina de seleção específica àquele tipo (figura 4.5). Esta rotina, por sua vez, verifica se o método é definido ou herdado (e não redefinido) pelo tipo em questão. Em caso afirmativo, é invocada a implementação do método. Caso contrário, é reportado erro de execução e o programa é interrompido.

A rotina de seleção de um tipo específico é composta basicamente por um comando de seleção com uma entrada para cada método aplicável ao tipo em questão, identificadas por rótulos inteiros consecutivos.

A função `Applicable` verifica se um determinado método é aplicável a um dado tipo e retorna a posição de sua respectiva entrada no *binder* (ou 0, se o método não é aplicável àquele tipo).

Para que esta verificação seja possível, conceitualmente é definida uma matriz bi-dimensional M tal que $M[t,m]$ corresponda à entrada do método de código m na rotina de *binding* do tipo de código t ou, em especial, 0 se o método não é aplicável ao tipo.

Esta matriz, entretanto, é, por natureza, bastante esparsa. Optou-se, portanto, por uma implementação alternativa que, embora menos eficiente, minimi-

[8] Com o auxílio de um otimizador de invocações pode-se tentar determinar o tipo do primeiro argumento em tempo de compilação. Neste caso, o *binding* é estático.

```
procedimento BINDER (Codigo_metodo: inteiro,Num_args: inteiro)
   caso RTS_get_top_minus_code(Num_args - 1) =
      1: _M_k_T_j_binder(Codigo_metodo);
      ...
      i: _M_k_T_l_binder(Codigo_metodo);
      ...
      n: _M_k_T_m_binder(Codigo_metodo);
   fim caso
fim procedimento
```

Figura 4.4: seleção do tipo.

zasse o gasto de memória. Assim, a cada tipo do programa é associada uma lista com os métodos que declara. A ordem dos métodos em cada lista corresponde a ordem de suas respectivas entradas na rotina de seleção do tipo em consideração. Para determinar se um método de código m é aplicável a um tipo de código t basta verificar se o método m está presente na lista do tipo t.

4.5 Geração de Código

Na fase de geração de código de um módulo m, a árvore de programa é percorrida e, a medida que construções sintáticas são visitadas, seus respectivos códigos são gerados. Ao final deste processo, resultarão os arquivos `m.h`, `m.cxx`, `m.lls` e, possivelmente, `RTS.cxx` (seção 4.3).

O arquivo `m.lls`, responsável pela propagação das informações necessárias ao compilador *LL* no momento de sua importação, possui a estrutura da figura 4.6.

A seção *$DECLARATIONS* contém declarações necessárias para a importação do módulo m. Sua sintaxe é bastante similar a de módulos ordinários, porém, corpos de construtores, métodos e expressões de inicialização de constantes são substituídos por *_NIL*; corpos de rotinas são ignorados.

Por razões descritas na seção 4.4, os métodos apenas podem ser enumerados no momento da compilação final do programa, assim, seus códigos são repre-

```
procedimento _M_T_binder (Codigo_metodo:  inteiro)

   I := Applicable(Codigo_metodo,_t_M_T_c)

   (*
      _t_M_T_c é o codigo do tipo T declarado em M.
   *)

   caso I =

      0:  Erro("Método não aplicável !")
      1:  (* invocacao do primeiro metodo de T *)
      ...
      n:  (* invocacao do n-ésimo metodo de T *)
      ...
      z:  (* invocacao do z-ésimo metodo de T *)

   fim caso

fim procedimento
```

Figura 4.5: seleção do método.

sentados por variáveis globais (posteriormente inicializadas). Porém, durante a compilação em separado de um módulo, muitas vezes não é possível determinar se este método é declarado em algum tipo do programa. Neste caso, embora utilizada, esta variável poderia não ser declarada, resultando em erro de compilação do código hospedeiro. A fim de evitar tal inconsistência, variáveis de códigos de métodos utilizadas na invocação do procedimento `BINDER` são listadas na seção *$USED_CODES* e iniciadas adequadamente. Caso o método não seja aplicável, é reportado erro de execução e o programa é interrompido elegantemente.

Cada módulo *m* possui em seu respectivo arquivo `m.cxx` procedimentos responsáveis pela seleção dinâmica de métodos de seus tipos (seção 4.4.2) e um procedimento (`_init_m`) com inicializações de suas constantes e com a tradução dos comandos do corpo do módulo.

O arquivo `m.h`, por sua vez, possui, basicamente, declarações de protótipos de funções e de variáveis que podem ser usadas pelo código hospedeiro de módulos que importem *m* ou pelo arquivo `RTS.cxx`.

```
(*$ BEGIN_LLS *)

$DECLARATIONS

module m is

   import m1,m2,...
   export n1,n2,...

   const C is_NIL;
   type T is
      constructors
         |New (x1,...)      ⇒ _NIL
      ...
   endtype

   procedure P (x1,x2,...) is
   endprocedure;

endmodule

$USED_CODES
   m_f2
$END

(*$ END_LLS *)
```

Figura 4.6: arquivo m.lls.

Finalmente, o arquivo `RTS.cxx` é constituído por inicializações de variáveis de códigos de tipos e de métodos (seção 4.3), pela rotina `BINDER` e por um procedimento (`START_UP`) responsável pela invocação das rotinas de inicialização dos módulos integrantes do programa na ordem adequada e pelo tratamento de seus argumentos.

Os efeitos de declarações *LL* podem se estender por vários módulos e pelos vários arquivos de código hospedeiro. Assim, a seguir são discutidos detalhes de suas traduções. Por outro lado, em virtude de seus efeitos sobre o processo de geração de código ser local ao corpo da rotina ou método onde estão inseridos, a tradução de comandos é apenas exemplificada pelo algoritmo de geração do comando **forall**. Detalhes acerca de outros comandos podem ser encontrados em

[BK94].

Vale observar que nomes usados no programa $\mathcal{LL}$ são codificados na linguagem hospedeira a fim de evitar ambigüidades.

4.5.1 Cláusulas de importação

from *M1* **import** ...;
import *M2*;

São inseridos no arquivo `M.cxx` diretivas "`includes`" para os arquivos `.h` de todos módulos importados por *M*, pois, naturalmente, este arquivo deverá ter acesso às entidades importadas por ele.

```
#include "M1.h"
#include "M2.h"
```

Na compilação final é necessário que sejam enumerados todos os tipos e métodos do programa. É, portanto, natural que seja necessário conhecer todos os módulos que integram o programa. Para que a informação dos módulos importados pelo módulo em compilação não seja extraviada, é inserida a declaração **import** `M1,M2,...` na seção *$DECLARATIONS* do arquivo `M.lls`.

4.5.2 Cláusulas de exportação

A declaração de entidades exportadas é colocada na seção *$DECLARATIONS* do arquivo `.lls` correspondente. Corpos de métodos e expressões que inicializam constantes são substituídos por *_NIL*; corpos de rotinas são desconsiderados. Note que, se um procedimento ou função `R` for exportado, todas as declarações de `R` são colocadas na seção *$DECLARATIONS*. A própria cláusula de exportação é copiada nesta seção, a fim de ser possível verificar se os nomes importados foram realmente exportados.

4.5.3 Constantes

const *C* **is** *expr*

Esta declaração $\mathcal{LL}$ vai gerar a declaração de variável

```
LLtype  _M_C_c;
```

no arquivo `M.cxx`. Se a constante for exportada, também será acrescida ao arquivo `M.h` a declaração

```
extern LLtype  _M_C_c;
```

Dentro do procedimento `_init_M` é gerado código para a avaliação da expressão *expr* e o resultado é atribuído à variável `_M_C_c`. Na seção de declarações do arquivo `M.lls` é gerado

```
const C is _NIL;
```

Se for possível determinar o tipo da expressão, é acrescentado na declaração acima a restrição adequada.

4.5.4 Tipos

type *T* **subtypeof** *T1* **is**
  constructors
    |*Ident(Args)* ⇒ *RoutineBody*
  procedures
    |*Ident(MethArgs)* ⇒ *RoutineBody*
  functions
    |*Ident(MethArgs)[Restricao]* ⇒ *RoutineBody*
  operators
    |*OpDecl* ⇒ *RoutineBody*
endtype;

A implementação de construtores e métodos (`RoutineBody`) foi encapsulada por procedimentos e funções *LL*. Uma invocação a métodos ou construtores será convertida em uma invocação destes procedimentos e funções. A geração de código para estas implementações serão feitas no momento da tradução daqueles procedimentos e funções.

A geração de código para uma declaração de tipo pressupõe várias tarefas:

- criação de uma variável global C++ que armazene o código do tipo;
- criação da constante *LL* do tipo *typetag* que denota o tipo;
- criação de variáveis globais C++ para os códigos dos métodos;
- geração do *binder* para o tipo (`_M_T_binder`, neste caso);
- atualização das estruturas de dados utilizadas pela função `Applicable` (seção 4.4.2).

Como a *variável do código do tipo* deve ser visível ao procedimento de *binding* do tipo *T* (`_M_T_binder`) para a invocação da função `Applicable` (veja seção 4.4.2) e como códigos de tipos exportados podem ser usados em comparações entre tipos, é inserida no arquivo `M.h` a declaração:

```
extern int _t_M_T_c
```

A inicialização desta variável ocorrerá no arquivo `RTS.cxx`.

A *criação da constante LL* do tipo *typetag* provoca a declaração

```
LLtype   _M_T_ct;
```

no arquivo `M.cxx`. Caso o tipo T seja exportado, a declaração

```
extern LLtype   _M_T_ct;
```

é incluída em `M.h`.

No procedimento `_init_M` é inserido código para a inicialização dos campos do objeto *LL* do tipo *typetag*, referente ao tipo *T*. Esta inicialização deve ser feita através da pseudo-função `TypeTag`, declarada no módulo *System*.[9]

As *variáveis de código de métodos* serão declaradas e inicializadas no arquivo `RTS.cxx` por declarações da forma:

[9] A árvore de hierarquia é construída pela pseudo-função `TypeTag`. Lembre-se que a análise semântica já verificou a hierarquia dos tipos.

```
int _<nome>_p<n>c = <C1>; /* p/ métodos-procedimentais */
int _<nome>_f<n>c = <C2>; /* p/ métodos-funcionais */
```

```
                    <nome>:  nome do método
                       <n>:  aridade do método
<C1>  e <C2>:  código do método
```

O *binder* para o tipo T (denominado `_M_T_binder`), gerado no arquivo `M.cxx`, tem seu algoritmo descrito na seção 4.4.2. É constituído, basicamente, de um comando de seleção, no qual cada rótulo é responsável pela invocação de um método do tipo, ou seja, para cada método do tipo T existe uma entrada no comando de seleção responsável por invocá-lo. É neste ponto que o processo de *binding* se efetiva.

As rotinas de *binding* de todos os tipos T declarados num módulo M devem ser "exportadas" para o arquivo `RTS.cxx` para serem usadas pela rotina `BINDER`, portanto, seus protótipos devem ser declarados no arquivo `M.h`.

Na seção *$DECLARATIONS* do módulo `M.lls` é colocada uma cópia da declaração do tipo cujos corpos de métodos e construtores são substituídos por *_NIL*. Note que mesmo que o tipo não seja exportado, é necessário colocar sua declaração no arquivo `.lls`, pois um objeto deste tipo pode ser retornado por alguma rotina ou método. O `BINDER` deve ser capaz de determinar o *binding* na invocação de métodos deste tipo.[10]

4.5.5 Procedimento e Funções

procedure P *(*$x_1,\ldots,x_n$*)* **is** ... **endprocedure**
function F *(*$y_1,\ldots,y_m$*) [Restrição]* **is** ... **endfunction**

Procedimentos e funções $\mathcal{LL}$ são traduzidos em procedimentos da linguagem hospedeira, pois, no caso de função, a própria tradução se encarregará de colocar o resultado na pilha de execução.

O código gerado para P e para F são colocados no arquivo `M.cxx` e serão da forma:

[10] Tipicamente isto ocorre com iteradores.

```
void _M_P_pn () {...}

void _M_F_fm () {...}
```

Os comandos *LC* contidos no procedimento ou na função são traduzidos e colocados nos respectivos procedimentos.

Restrições de tipos sobre argumentos provocarão a geração de código que verifique, em tempo de execução, se os argumentos satisfazem as restrições que lhes são impostas. Raciocínio análogo se aplica a restrições sobre valores retornados por funções.

Caso a rotina seja exportada, o protótipo do procedimento gerado na linguagem hospedeira são incluídos em `M.h` e sua declaração *LC* (com o corpo de comandos vazio) é inserida na seção *$DECLARATIONS* do arquivo `M.lls`

A tradução de uma invocação a uma sub-rotina compreende três passos. O *primeiro* é responsável pela avaliação dos argumentos da invocação. Os objetos resultantes são colocados na pilha de execução. O *segundo* consiste na tentativa de estaticamente determinar a declaração da rotina invocada, seja ela um procedimento, uma função, ou mesmo um método. Por fim, a invocação é efetivada. Caso, no passo anterior, tenha sido possível selecionar estaticamente a implementação da sub-rotina invocada ou por ela se referir a um procedimento (ou função), ou a um método qualificado, ou, ainda, se o otimizador de invocação [dC] tenha determinado o tipo do primeiro argumento do método, a invocação adequada é gerada. Caso contrário, tal invocação refere-se a algum método. Portanto, faz-se necessário que a seleção seja feita em tempo de execução pelo procedimento `BINDER`.

A seguir são exemplificadas as traduções de invocações a um procedimento P com um argumento (seleção estática) e a um método P com dois parâmetros (seleção dinâmica).

```
/* P(10); */

IntegerDenotation("10");
_M_P_p1();

/* P(1,2); */

IntegerDenotation("1");
IntegerDenotation("2");
BINDER(_P_p2c,2);
```

4.5.6 Pseudofunções

Pseudofunções diferem de funções ordinárias $\mathcal{LL}$ pelo fato de seus argumentos necessitarem de tratamento especial. A maioria delas, as *pseudofunções denotacionais*, têm por tarefa interpretar denotações do programa. Estas, que incluem *IntegerDenotation*, *RealDenotation*, *StringDenotation*, *BraceDenotation*, *BracketDenotation* e *AngleBracketDenotation*, não podem ser invocadas pelo programador $\mathcal{LL}$. São de uso exclusivo do compilador. A pseudofunção *TypeTag*, embora não seja responsável pela interpretação de denotações, é incluída neste grupo por sua utilização também ser interna.

Quaisquer outras pseudofunções, em especial *ProcRef* e *FuncRef*, são classificadas como *pseudofunções gerais* e possuem tratamento bastante semelhante ao dispensado a funções ordinárias, podendo, inclusive, ser invocadas num programa $\mathcal{LL}$, exportadas ou importadas entre módulos. Portanto, suas declarações provocam a inclusão dos protótipos das rotinas que as implementam no arquivo `System.h`. Caso exportadas, suas declarações são também inseridas em `System.lls`.

A invocação de pseudofunções gerais compreende o tratamento dos argumentos que, neste caso não resultarão em objetos $\mathcal{LL}$, mas sim valores de tipos conhecidos internamente ao compilador, e a geração da invocação da rotina que a implementa.

A declaração de pseudofunções denotacionais, da forma:

```
(*$ BEGIN_PSEUDOFUNCTIONS *)

    NomePsd (str)      == ImplPsd

(*$ END_PSEUDOFUNCTIONS *)
```

provoca a criação de diretiva

```
#define NomePsd ImplPsd
```

no arquivo `System.h`.

Pseudofunções denotacionais são invocadas pelo compilador sempre que uma denotação seja utilizada, porém, seu nome é utilizado em lugar do nome da implementação como no caso de pseudofunções gerais.

Esta diferença de tratamento se deve ao fato de que pseudofunções denotationais devam ser conhecidas globalmente ao programa, entretanto, não devem ser invocadas pelo programador. Isto impede a utilização do mecanismo de importação e exportação. Assim compilações em separado de outros módulos não teriam conhecimento do nome da implementação destas pseudofunções.

A seguir é mostrado um exemplo da tradução de invocações a pseudofunções denotationais e gerais. As declarações e comandos $\mathcal{LL}$:

```
(*$ BEGIN_PSEUDOFUNCTIONS *)

    IntegerDenotation(str):integer==  implInt
    ProcRef(proc,int):procref      ==  implProcRef

(*$ END_PSEUDOFUNCTIONS *)
...
x ← 10;
ProcRef(P,2);
```

são traduzidos para:

```
#define IntegerDenotation implInt

...

IntegerDenotation("10");
x = RTS_pop();
implProcRef(_M_P_p2,2);
p = RTS_pop();
```

4.5.7 Corpo do módulo

Os comandos do corpo de um módulo *M* são traduzidos como se fizessem parte de um procedimento *ℒℒ*. O resultado da tradução é colocado no procedimento `_init_M`, no arquivo `M.cxx`, após o código das inicializações, já descritas, e seu protótipo é colocado em `M.h`, pois será invocado pela rotina `START_UP`.

4.5.8 Comando forall

O algoritmo de tradução do comando **forall**, descrito a seguir, permite exemplificar vários aspectos sobre a geração de código de comandos.

forall *var* **in** *expr* **do**
StatmentSequence
endforall

O primeiro aspecto evidencia a preocupação em informar ao programador a localização de um erro de execução. Caso o programa tenha sido compilado de forma a incluir facilidades de depuração, no início da tradução de cada comando é armazenado, em uma variável denominada `RTSpc`, o número da linha do programa onde o comando foi utilizado. Isto permite que a mensagem reportando a ocorrência de algum erro durante a execução do comando seja enriquecida com o número da linha indicado por `RTSpc`.

Nomes de escopo global (constantes, tipos, códigos de tipos e de método, etc) são codificados em sua tradução a fim de evitar ambigüidades, porém, nomes de variáveis não necessitam qualquer codificação, visto que seu escopo é unicamente local. Assim, `_M_T_Empty_f1` denota o método funcional `Empty`, declarado no tipo `T` e no módulo `M`, com um parâmetro formal, enquanto que `_Empty_f1c` armazena o código de um método funcional `Empty`, de aridade 1.

Uma terceira preocupação consistiu em antever possíveis otimizações de invocação a métodos. Embora tais otimizações estejam ainda em estudo [dC], o compilador foi estruturado de forma a, tão logo o otimizador esteja pronto, sua integração seja automática. Note-se que a geração de invocações a métodos sempre apresentam duas alternativas: ou é gerada uma invocação ao procedimento `BINDER` a fim de que a seleção do método seja dinâmica, ou, caso o otimizador tenha obtido sucesso na determinação do tipo do primeiro argumento, é gerada a invocação direta à tradução do método (seleção estática).

No algoritmo de tradução o código hospedeiro é apresentado em fonte *tt* e é inserido no arquivo adequado através do procedimento `Insira`. Números são convertidos em cadeias de caracteres pela função `STR`. Cadeias de caracteres podem ser concatenadas pela utilização do operador `+`.

```
algoritmo Forall

    se Depuração
       então Insira("RTSpc = " + STR(num_linha) + ";")
    fim se

    Gere código para expr
    Insira("for( ; ; ) { ")
    Insira("RTS_push(" + NomeConstante("System","true") + ");")
    Insira("RTS_dup_minus(1);")

    se Otimizador determinou tipo do iterador
       então Insira("_M_T_Empty_f1();")
       senão Insira("BINDER(_Empty_f1c,1);")
    fim se

    (* O resultado da aplicação do método Empty
    do iterador está no topo da pilha. *)

    Insira(NomeMetodoFunc("System","boolean","OperatorEqual",2) + ;)
    Insira("if (RTS_pop() == " + NomeConstante("System","true") + ")
    break;")
    Insira("RTS_dup();")

    se Otimizador determinou tipo do iterador
       então Insira("_M_T_Next_f1();")
       senão Insira("BINDER(_Next_f1c,1);")
    fim se

    (* Resultado da aplicação do método Next
    do iterador está no topo da pilha. *)

    se Existe restrição de tipo sobre var
       então Gere código para verificação
    fim se

    Insira("var = RTS_pop();")
    Gere código para StatmentSequence
    Insira("}")
    Insira("RTS_pop();")

fim algoritmo
```

Capítulo 5

Exemplo

Neste capítulo é apresentado um problema e descrito informalmente um algoritmo para sua solução em um grau de abstração adequado a discussões acadêmicas. São também providas duas implementações distintas do mesmo algoritmo.

O problema da visibilidade de um polígono a partir de um ponto pode ser descrito como:

> *Seja um polígono simples P, representando as paredes de um recinto fechado. Considere um ponto x, interno a P, que representa uma fonte pontual de luz (figura 5.1). Deseja-se determinar um polígono V, representado pela seqüência ordenada de seus vértices, que delimite a região iluminada pela fonte de luz (figura 5.2).*

5.1 Algoritmo

O algoritmo é baseado em [JS85] e apresentado em [O'R87]. Sua descrição é bastante intuitiva. Não é necessário, neste momento, apresentá-lo em grandes detalhes, mas, apenas, deseja-se fornecer subsídios para o estabelecimento de paralelos entre as implementações.

Basicamente o algoritmo consiste em, partindo de um vértice v_0, percorrer a fronteira de P no sentido anti-horário, construindo o polígono de visibilidade V. Durante o percurso podem ser encontradas regiões "escuras" de P. Estas regiões não devem integrar o polígono V, implicando, inclusive, em possíveis reconstruções.

Considere os seguintes casos, nos quais $v_0, v_1, \ldots, v_i$ pertencem ao polígono V e deseja-se determinar o seu estado após a análise de v_{i+1}.

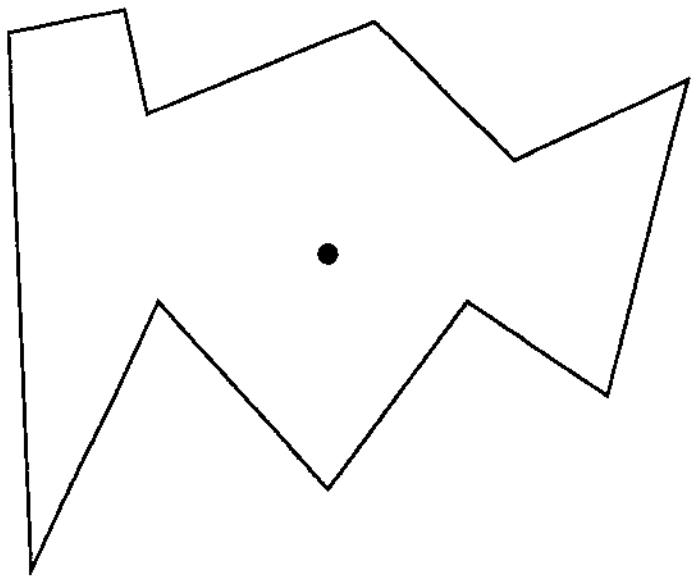

Figura 5.1: polígono e ponto interno.

Na figura 5.3 a aresta v_iv_{i+1} encobre parte do que se julgava "iluminado" por x. Neste caso alguns vértices de V devem ser eliminados (desempilhados) até que se encontre algum ponto v_j visível a partir de x. A eliminação adequada dos vértices é desempenhada pelo procedimento POP do algoritmo (figura 5.5).

A figura 5.4 ilustra o caso quando a aresta v_iv_{i+1} é encoberta por V. A partir deste momento, v_{i+1} e alguns de seus sucessores (em P) são desconsiderados até que se encontre um ponto v_j, a partir do qual a fronteira de P seja novamente "iluminada" (procedimento WAIT).

Se nenhum dos dois casos se verifica, o vertice v_{i+1} é acrescentado ao polígono V através do procedimento PUSH.

A função Virada_a_direita(x,p_1,p_2) é verdadeira quando o ponto x está à direita da reta determinada por p_1 e p_2, orientada de p_1 para p_2.

5.2 Implementações

As implementações mostradas nas figuras 5.6 e 5.7 tentam ser homogêneas, no sentido de que ambas desfrutam do mesmo nível de abstração. As implementações das rotinas auxiliares são omitidas por brevidade.

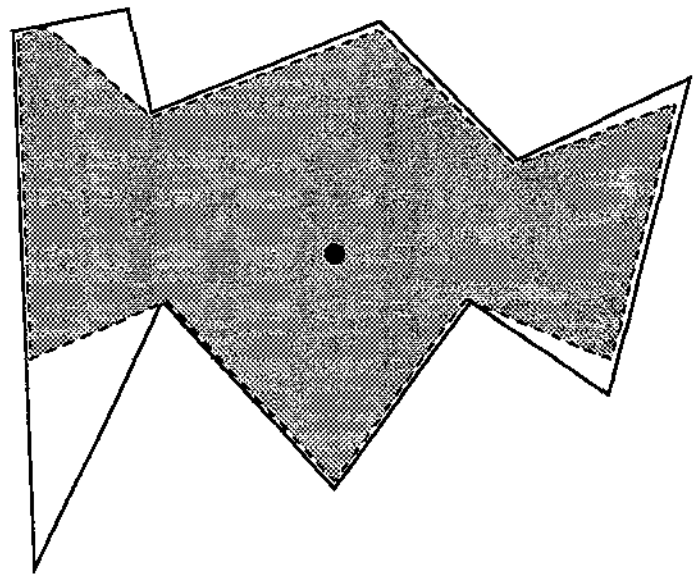

Figura 5.2: polígono de visibilidade.

5.3 Comparação

Ambas implementações derivam diretamente da descrição do algoritmo em alto nível. Conseqüentemente, como era de se esperar, as suas estruturas são muito semelhantes. As principais diferenças estão reduzidas, na realidade, à sintaxe. A implementação em $\mathcal{LL}$ é ligeiramente mais compacta devido à ausência de declarações que não são obrigatórias.

As duas implementações dependem da existência de bibliotecas de rotinas adequadas à solução do problema em questão. O exemplo ilustra que, neste caso, a linguagem $\mathcal{LL}$ parece ser tão expressiva quanto C++. Deve-se lembrar que o objetivo principal da linguagem $\mathcal{LL}$ é justamente prover uma ferramenta para programação neste nível.

Uma conclusão óbvia poderia ser a de que não vale a pena o esforço de implementação de uma nova linguagem para conseguir tão pouco! Por outro lado, o exemplo demonstra algumas diferenças que parecem tornar $\mathcal{LL}$ mais "natural" do que C++. Por exemplo, a expressão

RightTurn(SuccTop(), Top(S),Inf(vert_list,v))

parece ser mais legível do que a correspondente

(P[v→right_turn (*Pilha [Pilha.succ (topo)], *Pilha [topo]))

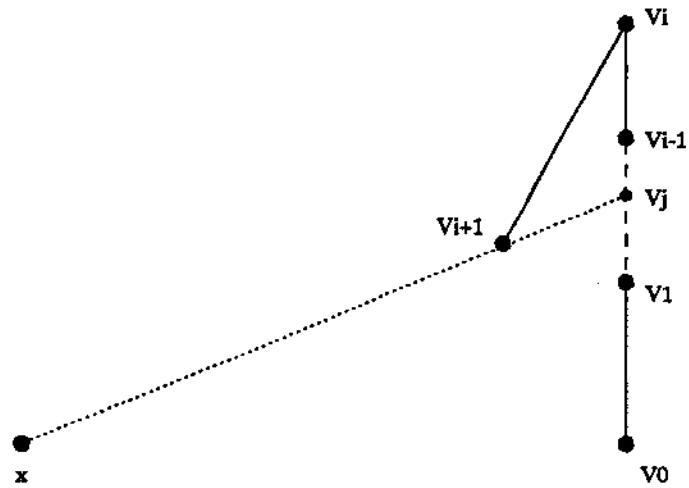

Figura 5.3: polígono de visibilidade é encoberto.

e também mais semelhante à descrição do algoritmo em alto nível. A necessidade de utilizar em C++ os símbolos "→" e "*" indica que o programador precisou preocupar-se com detalhes de implementação das estruturas de dados e não apenas com o algoritmo, diminuindo, de uma certa maneira, o grau de abstração.

Mesmo assim, poderia-se argumentar que basta utilizar um subconjunto "bem comportado" de C++ para conseguir os mesmos objetivos. Entretanto, é um fato notório que C++ é uma linguagem muito complexa, com vários mecanismos cuja compreensão exata exige bastante especialização. Assim, é muito comum cometer-se enganos de programação que escapam ao subconjunto bem comportado, mas não à linguagem como um todo. Um dos objetivos do projeto de ℒℒ foi justamente a simplicidade que diminuísse a possibilidade deste tipo de erros.

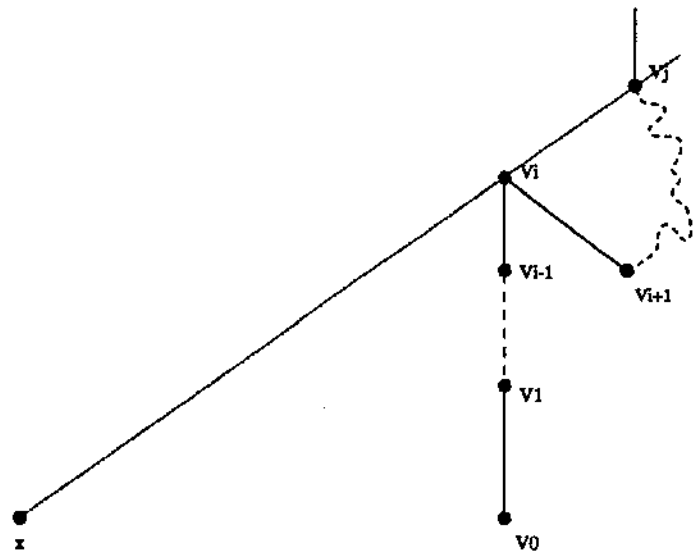

Figura 5.4: aresta encoberta pelo polígono de visibilidade.

```
algoritmo Visibilidade

   Determine v0

   PUSH(v0)
   PUSH(v1)

   para todo vi de P (i ≥ 2) faça

      se Virada_a_Direita(x,vi-1,vi)
         entao (* algo foi escondido *)
            se Virada_a_Direita(vi,vi-2,vi-1)
               entao WAIT() (* V esconde vi *)
               senao POP() (* vi-1vi esconde V *)
            fim se
         senao PUSH(vi)
      fim se
   fim para
fim algoritmo
```

Figura 5.5: algoritmo.

```
function PointVisibilityPolygon(P;x) is[a]

    vert_list := Vertices(P);
    ini := Determine_Inicio(vert_list,x);
    v := CyclicPred(vert_list,ini);

    Push(S,Inf(vert_list,ini));
    Push(S,Inf(vert_list,v));

    while (Inf(vert_list,v) <> Inf(vert_list,ini)) do
       v := CyclicPred(vert_list,v);
       if RightTurn(Top(S),Inf(vert_list,v),x)
          then
             if RightTurn(SuccTop(), Top(S),Inf(vert_list,v))
                then
                   pto_dir := ponto_a_direita(x,Top(S));
                   janela := SetJanela(Top(S),pto_dir);
                   v := Wait(vert_list,v,janela,x);
                else Pop(vert_list,v,x);
             endif;
          elsePush(x,ini,vert_list,v);
       endif;
    endwhile;

endfunction;
```

[a]Implementado por Cristina Célia de Melo Barros.

Figura 5.6: versão $\mathcal{LL}$.

```
void PointVisibility::vis_ponto (Point2D x,
                                 list_item ini, list_item fim,
                                 EnSentido sentido = NA_MAO)
{
   list_item v,vaux;
   Point2D pto_dir,p1,p2,p3;
   Ray2D Janela;

   topo = Pilha.push(P [ini]);
   v = Proximo(ini);
   topo = Pilha.push(P [v]);
   while (v != fim)
   {
      v = Proximo(v);
      if (x.right_turn(*Pilha[topo],*P[v]) )
      {
         if (P[v]→right_turn(*Pilha [Pilha.succ(topo)], *Pilha[topo]))
         {
            pto_dir = ponto_a_direita(x,*Pilha [topo]);
            Janela.set(*P [topo],pto_dir);
            WAIT(v,Janela);
         }
         else POP(v);
      }
      else PUSH(v);
   }
}
```

Figura 5.7: versão C++.

Capítulo 6

Conclusões e trabalhos futuros

Foi apresentada uma ferramenta (linguagem de programação e compilador) para um público específico que permite implementação rápida e segura de algoritmos. O tempo dispendido no projeto de uma nova linguagem e na implementação de seu compilador é justificado pela crença que problemas específicos são melhor resolvidos por ferramentas particulares, evitando o trabalho, por vezes árduo, de compatibilização do modelo conceitual da ferramenta às particularidades do problema. O capítulo 5 apresenta um exemplo típico para ilustrar esta tese.

Além disso, $\mathcal{LL}$ se encaixa em um nicho pouco explorado: por um lado distingue-se da maioria das linguagens de programação tradicionais por ser voltada a um grupo de usuários bastante particular (projetistas de algoritmos), buscando simplicidade e economia de conceitos, o que a aproxima de *little languages* (`AWK`, `PERL`, etc), porém destina-se às mais diversas aplicações, o que simultaneamente a re-aproxima de linguagens de programação de sistemas e a distancia de *little languages*.

A linguagem apresenta uma forma de reutilização de bibliotecas dentro de um ambiente amistoso e seguro. A discussão deste mecanismo foi enriquecida com a exposição de algumas dificuldades encontradas no processo de integração de bibliotecas externas ao ambiente $\mathcal{LL}$, culminando na apresentação de atualizações que possivelmente podem suavizar os inconvenientes descritos. Uma destas sugestões, a extensão de tipos, talvez deva ser seriamente considerada em futuras atualizações da linguagem.

O trabalho também abriu caminho para novas linhas de pesquisa, notadamente, o otimizador de invocações [dC], assunto de dissertação de mestrado em andamento.

Atualmente encontra-se em fase final o interfaceamento da biblioteca Leda

[PB, Nah]. É natural desejar que outras bibliotecas sejam compatibilizadas ao sistema, sobretudo aquelas que permitam o desenvolvimento de aplicações orientadas a ambientes de janelas. Seria bastante útil que estas bibliotecas fossem independentes de padrões específicos (ou ao menos facilmente adaptáveis), facilitando a migração de aplicações de $\mathcal{LL}$ para outras plataformas. Bibliotecas estatísticas, matemáticas e de manipulação de processos também seriam úteis ao ambiente.

Um segundo trabalho consistiria na implementação de outros módulos de geração que produzissem, por exemplo, código `Modula-3` ou, ainda, código em linguagem de montagem. Com relação a última opção, devem ser identificadas as dificuldades de integração de bibliotecas externas ao ambiente, uma vez que o código gerado pelo compilador estaria um nível abaixo da camada de ligação entre a biblioteca e a aplicação $\mathcal{LL}$.

Como terceira proposta poderia ser citada a implementação de um gerador de *makefiles* para compilação de programas $\mathcal{LL}$. O protótipo do compilador tenta ao máximo estar desvinculado do ambiente que o circunda para permitir a maior flexibilidade possível em termos de linguagens hospedeiras. Para tanto o compilador $\mathcal{LL}$ não se preocupa com a invocação adequada ao compilador hospedeiro. É quase imprescindível, portanto, a implementação de um aplicativo responsável pelo gerenciamento da compilação completa de um programa $\mathcal{LL}$.

Ainda poderia ser citada a implementação de um ambiente de programação composto por um interpretador de comandos $\mathcal{LL}$ e por um depurador de programas. Ambos poderiam trabalhar em cooperação com um ambiente como o `Geolab` auxiliando-o na composição de algoritmos. Por outro lado, o ambiente `Geolab` permite que algoritmos geométricos escritos em `C++` sejam ligados e ativados por seu intermédio através de um protocolo de conversação. Este protocolo poderia ser adaptado a fim de `Geolab` ter a capacidade de conversar com programas $\mathcal{LL}$.

$\mathcal{LL}$ parece ser facilmente interfaciável a bibliotecas de linguagens imperativas. Seriam úteis interfaces com linguagens funcionais ou lógicas? Quais seriam as dificuldades de fazê-lo?

A programação de interfaces propriamente dita consiste basicamente de quatro passos:

1. pela inspeção da assinatura da rotina (ou do método) identifica-se o número de argumentos da invocação e as restrições de tipos que devem ser impostas a cada um dos argumentos. A partir desta inspeção são codificadas invocações adequadas ao RTS para que argumentos sejam retirados da pilha de execução e atribuídos ao parâmetros formais, fazendo verificações de tipos quando necessário;

2. a representação interna dos argumentos é "desempacotada";

3. é implementado o comportamento da sub-rotina através de invocações a operações da biblioteca hospedeira;

4. o resultado da operação é "empacotado" dentro de um objeto $\mathcal{LL}$, retornado através da pilha de execução.

Os dois primeiros passos são facilmente automatizáveis, o último passo demanda alguma intervenção do programador e, naturalmente, o terceiro item não pode ser automatizado. Seria possível, portanto, a implementação de ferramentas que codificassem parcialmente as rotinas de interfaceamento. Seria, inclusive, adequada a implementação de uma ferramenta deste tipo para cada possível linguagem hospedeira.

A linguagem $\mathcal{LL}$ não apresenta meios de gerenciamento automático de memória (coleta de lixo), o que talvez possa ser considerado como deficiência. A inclusão de tal mecanismo, entretanto, apresenta alguns obstáculos, uma vez que a implementação externa de tipos pode ser bastante dificultada se o programador de interfaces precisar se preocupar com este detalhe. Por outro lado, parecem promissoras investigações que tenham por objetivo definir mecanismos simples de gerenciamento automático de memória para programas $\mathcal{LL}$.

Uma primeira idéia neste sentido seria acrescentar métodos à classe `LLtype` do sistema de execução de $\mathcal{LL}$ de forma que, sempre que possível, fossem invocados métodos $\mathcal{LL}$ adequados para desalocação de um objeto específico.

Por fim, a linguagem $\mathcal{LL}$ é passível de atualizações. Algumas delas já foram sugeridas (múltiplas atribuições, extensão de tipos, cadeias de implementações apenas no topo da hierarquia de tipos). Poderia ser ainda considerada a adição de *macros sintáticas* à linguagem a fim de incrementar a expressividade e a legibilidade dos programas, permitindo a usuários ou projetistas de interfaces criassem novas estruturas sintáticas para $\mathcal{LL}$ quando considerassem necessário. Os comandos **while**, **repeat** e **forall**, por exemplo, poderiam ser definidos por macros. Um outra alternativa dentro desta vertente poderia ser investigada. Seja, por exemplo, a intenção de se prover uma estrutura sintática para descrição de corotinas em programas $\mathcal{LL}$. Tal estrutura deveria ser muito bem conhecida pelo compilador a fim de que fosse possível que o código correspondente fosse gerado? Seria possível definir e implementar um mecanismo simples e claro que permitisse a expressão deste tipo de estrutura por meio de macros sintáticas?

Bibliografia

[B+89] G.S. Blair et alli. Generecity vs Inheritance vs Delegation vs Conformance vs ... *Jornal of Object Oriented Programming*, pp. 1–19, 1989.

[BK94] E. Bacarin e T. Kowaltowski. Relatório de Implementação do Compilador $\mathcal{LL}$, 1994. DCC, UNICAMP.

[Bro90] B. Brown. Non-Linear Type Extensions. *ACM SigPlan Notices*, 29(2), 1990.

[CW85] L. Cardelli e P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[dC] C. R. da Costa. Otimização de Tipos em Linguagem $\mathcal{LL}$. Disserta,c ao de Mestrado em andamento, UNICAMP.

[EF+91] S. Engelstad, K. Falck, et alli. A Dynamic C-Based Object-Oriented System for Unix. *IEEE Software*, pp. 73–85, maio de 1991.

[F+94] L. H. Figueiredo et alli. The design and implementation of a language for extending applications. In *Anais do XXI Seminário Integrado de Software e Hardware*, pp. 273–283, 1994.

[Fle93] J. Fleming. The C+@ Programming Language. *Dr. Dobbs*, pp. 24–32, outubro de 1993.

[GR83] A. Goldberg e D. Robson. *Smalltalk-80: The language and its Implementation.* Addison-Wesley, 1983.

[Jac] W. R. Jacometti. Geolab – Um Ambiente para Desenvolvimento de Algoritmos em Geometria Computacional. Disserta,c ao de Mestrado, DCC, UNICAMP, 1992.

[JS85] B. Joe e R.B. Simpson. Visibility of a simple polygon from a point. Technical report, Univ. Waterloo, 1985.

[JW88] K. Jensen e N. Wirth. *Pascal ISO - Manual do Usuário e Relatório.* Editora Campus, 1988.

[KB93] T. Kowaltowski e E. Bacarin. *LL*– An Object-Based Library Language, Reference Manual. Technical report, DCC, Unicamp, 1993.

[L+79] B. Liskov et alli. *CLU Reference Manual.* Massachusetts Institute of Technology, 1979.

[MW92] H. Mossenbock e N. Wirth. The Programming Language Oberon-2. Technical report, Institut fur Computersysteme, jan de 1992.

[Nah] S. Naher. *Leda User Manual Version 3.0.* Max-Plank-Institut fur Informatik.

[Nel91] G. Nelson. *System Programming with Modula-3.* Prentice-Hall, 1991.

[O'R87] J. O'Rourke. *Art Gallery Theorems and Algorithms.* Oxford University Press, 1987. pp. 203-206.

[Pax] V. Paxson. flexdoc (man page).

[PB] A.C.M. Paquola e C.C.M. Barros. Implementação de interfaces para linguagem *LL*. Relatório de Atividades. DCC, UNICAMP, 1994.

[S+85] C. Schaffert et alli. *Trellis Object-based Environment - Language Reference Manual.* Eastern Research Lab - Digital Equipment Corporation, 1985.

[Sta] R. Stallman. Bison (infofile).

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Weg87] P. Wegner. Dimensions of Object-Based Language Design. oct de 1987.

[Wir82] N. Wirth. *Programming in Modula-2.* Spring-Verlag, 1982.

[Wir88a] N. Wirth. The Programming Language Oberon. *Software - Practice and Experience*, pp. 671–690, jul de 1988.

[Wir88b] N. Wirth. Type Extensions. *ACM Transactions on Programming Languages and Systems*, 10(2):204–214, 1988.