Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Compilação de um programa escrito em linguagem qualquer para Haskell

Fernando Magno Quintão Pereira
Orientador: Roberto da Silva Bigonha
Co-Orientadora: Mariza A. S. Bigonha

Relatório Técnico do
Laboratório de Linguagens de Programação
LLP002/2001

Av. Antônio Carlos, 6627
31270-010 - Belo Horizonte - MG
26 de Agosto de 2003

Conteúdo

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho contém a descrição de uma ferramenta capaz de gerar interpretadores para linguagens de programação a partir da descrição dessas linguagens via programas $\mathcal{SCRIPT}$. $\mathcal{SCRIPT}$ é uma linguagem funcional cujo propósito principal é prover uma notação adequada para a descrição dos aspectos léxicos, sintáticos e semânticos de linguagens de programação. Os interpretadores automaticamente produzidos por esta ferramenta são programas codificados em $\mathcal{HASKELL}$, uma outra linguagem pertencente ao paradigma funcional.

Palavras Chaves: Interpretador, Script, Haskell, Linguagem Funcional, Tradutor, Geração Automática de Código, Semântica Denotacional

1 Introdução

O trabalho apresentado neste relatório faz parte de um projeto maior, que envolve professores e estudantes pesquisadores do Laboratório de Linguagens de Programação da UFMG e que se desenvolve em torno da linguagem $\mathcal{SCRIPT}$ [2].

O objetivo deste projeto é desenvolver uma ferramenta capaz de gerar automaticamente interpretadores para linguagens de programação a partir das descrições léxicas, sintáticas e semânticas das mesmas. Neste contexto, pretende-se que $\mathcal{SCRIPT}$ seja utilizada para descrever os três aspectos de uma linguagem de programação e que o interpretador produzido automaticamente seja um programa escrito na linguagem $\mathcal{HASKELL}$.

Entre outros trabalhos relacionados à Linguagem $\mathcal{SCRIPT}$ pode-se citar:

- O projeto e implementação de um compilador de $\mathcal{SCRIPT}$ para $\mathcal{LAMB}$, uma versão estendida do cálculo λ, a linguagem utilizada como código intermediário na compilação de linguagens funcionais. [3].
- Um tradutor de definições de funções escritas em $\mathcal{SCRIPT}$ para $\mathcal{HASKELL}$ [4].

O restante deste documento contém uma breve explicação sobre as atividades relacionadas ao projeto do Gerador de Interpretadores desenvolvidas no decorrer do primeiro semestre de 2001. As demais seções estão organizadas da seguinte forma:

- na Seção 2 é apresentado um resumo de todo o processo de geração automática de interpretadores e de interpretação de programas que está sendo desenvolvido;
- na Seção 3 é descrita a linguagem $\mathcal{SCRIPT}$ e são apresentados alguns exemplos de descrição de linguagens de programação;
- a Seção 4 contém uma breve descrição da Linguagem $\mathcal{HASKELL}$ e de seu papel neste projeto;
- na Seção 5 é tratada a questão da entrada de dados para os interpretadores gerados automaticamente e é dada uma breve visão de uma ferramenta de tradução desenvolvida como parte deste projeto;
- na Seção 6 é descrito um compilador de $\mathcal{SCRIPT}$ para a Linguagem $\mathcal{LAMB}$ e é mostrado como espera-se incorporar o código do Gerador de Interpretadores àquele compilador;

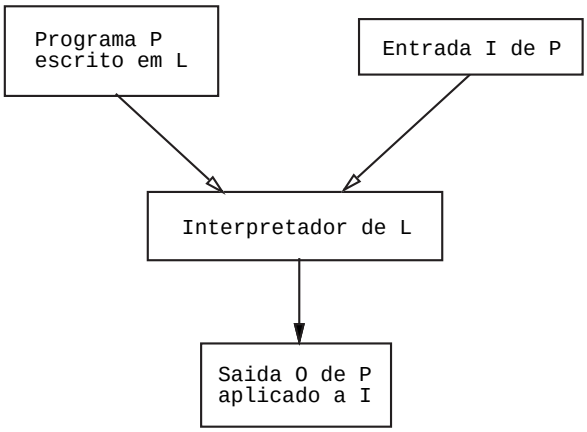

Figura 1: Visão geral do funcionamento de um interpretador

- a Seção 7 contém um breve relato de todas as atividades desenvolvidas no primeiro semestre de 2001 relacionadas a este Projeto Orientado;
- a Seção 8 contém uma descrição dos principais problemas que deverão ser solucionados na segunda fase do projeto;
- por fim, a Seção 9 apresenta as principais conclusões da primeira parte deste Projeto Orientado em Computação.

No contexto deste trabalho denomina-se $\mathcal{L}$ uma linguagem de programação de propósito geral. Um interpretador de $\mathcal{L}$ é um software capaz de receber um programa P_L, escrito em $\mathcal{L}$, mais os dados de entrada deste programa (D_{in}) e gerar, como saída, o resultado esperado da execução de P_L sobre D_{in}.

2 Visão Geral do Projeto Proposto

Um interpretador de programas funciona, de forma geral, de acordo com o esquema mostrado na Figura 1. O interpretador de $\mathcal{L}$ recebe duas entradas: o código fonte de um programa escrito em $\mathcal{L}$ e a entrada que normalmente seria oferecida à versão executável deste programa. O resultado obtido via interpretação é exatamente o mesmo que seria obtido caso o programa escrito em $\mathcal{L}$ fosse executado sobre sua entrada.

O funcionamento dos interpretadores que pretende-se gerar automaticamente não é exatamente igual ao descrito pela Figura 1. Os interpretadores gerados automaticamente não vão receber como entrada o código fonte dos programas que deverão interpretar. Ao invés disso, eles receberão uma descrição da estrutura

sintática do programa, isto é, como cada componente da linguagem a ser interpretada se organiza para compor seu código.

Devido a esta peculiaridade, além de um Gerador de Interpretadores para uma linguagem $\mathcal{L}$ qualquer, deverá também ser projetado e implementado um tradutor, capaz de codificar programas escritos em $\mathcal{L}$ para a estrutura de dados adequada ao interpretador produzido automaticamente. Opcionalmente, pode-se desenvolver um ambiente no qual todo o processo, desde a geração de um interpretador até a posterior tradução de programas para uma forma de entrada adequada ao interpretador assim gerado, possa ser feito em um único passo.

A fim de simplificar o processo descrito, ele foi dividido em três etapas:

1. a geração do interpretador de $\mathcal{L}$, escrito na linguagem funcional $\mathcal{HASKELL}$ e a geração de um tradutor capaz de traduzir programas escritos em $\mathcal{L}$ para uma estrutura de dados adequada a $\mathcal{HASKELL}$.

2. A tradução de um programa P_L, escrito em $\mathcal{L}$ para uma estrutura da linguagem $\mathcal{HASKELL}$, própria para ser enviada como entrada para o interpretador gerado na Etapa 1.

3. A submissão da estrutura obtida na Etapa 2 ao interpretador gerado na Etapa 1.

Todo o processo descrito encontra-se resumido nos esquemas das Figuras 2, 3 e 4. Cada uma dessas figuras representa uma das três etapas do processo, que vai desde a geração de um interpretador para a linguagem $\mathcal{L}$ até a execução dos programas P_L, via este interpretador.

3 A Linguagem $\mathcal{SCRIPT}$

$\mathcal{SCRIPT}$ é uma linguagem funcional, com algumas características do paradigma de Orientação a Objetos. A linguagem $\mathcal{SCRIPT}$ foi idealizada pelo professor Roberto Bigonha [2] e seu propósito principal é descrever outras linguagens de programação. A partir da descrição $\mathcal{SCRIPT}$ de uma linguagem $\mathcal{L}$ é possível gerar interpretadores e analisadores léxicos e sintáticos para $\mathcal{L}$, sendo este o objetivo principal do projeto.

Algumas características importantes da linguagem $\mathcal{SCRIPT}$ estão enumeradas logo abaixo:

- Linguagem do paradigma funcional.

- Equivalência estrutural de tipos.

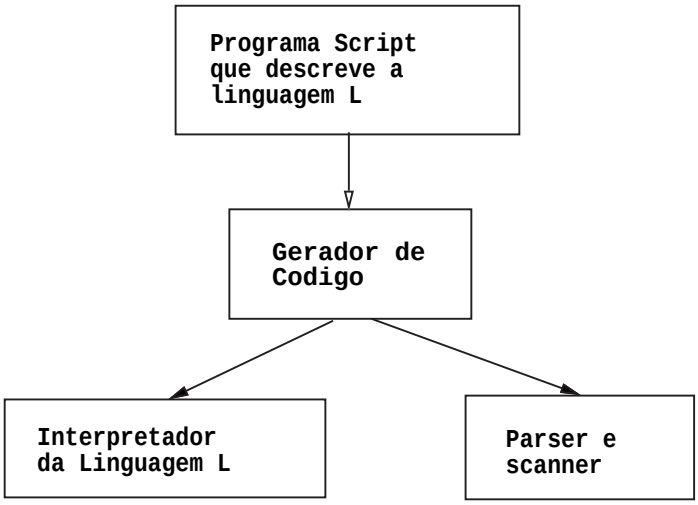

Figura 2: Primeira Etapa

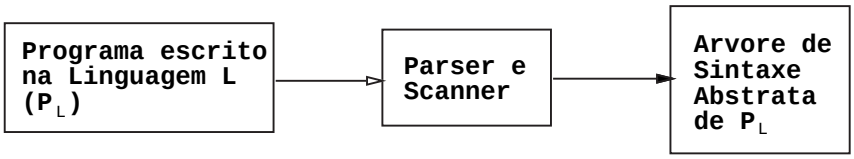

Figura 3: Segunda Etapa

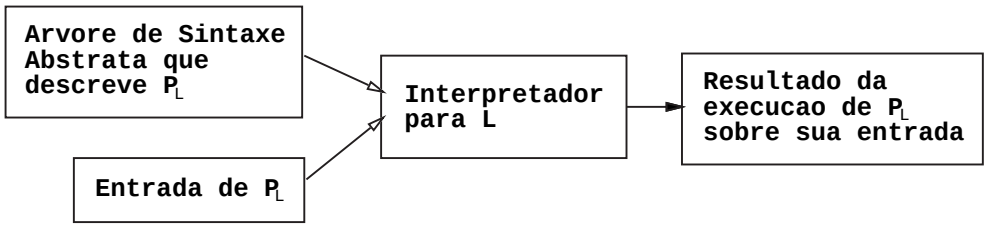

Figura 4: Terceira Etapa

```
GRAMMAR Arit
SYNTAX
  exp   ::=     exp "+" exp
        |       num
LEXIS
  UNIT ::= num
  num  ::= digit + : NUMBER;
  digit === "0" .. "9"
END Arit
```

Figura 5: Programa $\mathcal{SCRIPT}$ para somas de números inteiros.

- Linguagem altamente modular.
- Dotada de mecanismos de controle de visibilidade e encapsulação.
- Dotada de características de linguagens orientadas a objetos:
  - mecanismo de herança;
  - funções sobrecarregadas;
  - associação (binding) dinâmica.

Uma descrição $\mathcal{SCRIPT}$ completa engloba os três aspectos principais de uma linguagem: léxico, sintático e semântico. Dada a grande modularidade de $\mathcal{SCRIPT}$, cada um destes aspectos pode ser especificado em um arquivo diferente, ou no mesmo arquivo, ou ainda em vários arquivos separados.

A título de ilustração, a Figura 5 contém a descrição dos aspectos léxicos e sintáticos de uma linguagem muito simples, denominada `Arit`, capaz de reconhecer somas de números inteiros. A descrição dos símbolos reconhecidos por esta linguagem encontra-se na seção denominada `LEXIS` ao passo que a gramática que a define está especificada na seção identificada pelo rótulo `SYNTAX`. O conjunto de símbolos reconhecidos por essa linguagem é formado somente por seqüências de números inteiros e pelo sinal de adição (+).

Na seção `LEXIS` está definido o conjunto de símbolos reconhecido pela linguagem, ou seja, o seu alfabeto. Caso estes símbolos sejam agrupados em uma certa seqüência, podem vir a formar um programa correto. Os símbolos de uma linguagem formam um programa correto quando estão agrupados em uma ordem que concorda com a sua gramática. Toda linguagem de programação possui uma gramática bem definida que define quais seqüências de símbolos formam programas válidos. A gramática que define a estrutura sintática da linguagem `Arit` pode

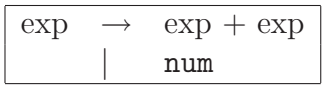

Figura 6: Gramática da Linguagem `Arit`

```
MODULE Arit

DOMAINS
    Exp = [Exp "+" Exp] | [N]

DEFINITIONS

DEF elab-exp (exp) : N =
        CASE exp
            /[exp1 "+" exp2] ->
                LET exp1' = elab_exp (exp1)
                LET exp2' = elab_exp (exp2)
                IN exp1' PLUS exp2'
            /[n] -> n
        END

END Arit
```

Figura 7: Descrição semântica da Linguagem `Arit`

ser vista na Figura 6. Observe que todas as regras de derivação da gramática aparecem no módulo `SYNTAX` do programa da Figura 5. Neste exemplo, a gramática da linguagem `Arit` possui somente duas regras de derivação.

Além da descrição dos aspectos léxico e sintático de uma linguagem, para que sua especificação esteja completa, é preciso que seja descrito também o significado de cada um de seus possíveis comandos e expressões. Este é o aspecto semântico de uma linguagem de programação. Tais descrições, em $\mathcal{SCRIPT}$ são feitas por meio de declarações de tipos e funções que descrevem as ações que serão realizadas por cada construção correta da linguagem. O programa da Figura 7 contém as funções semânticas que descrevem a linguagem `Arit`. Este programa formaliza o significado de cada construção da Linguagem `Arit`. Nesta linguagem, uma expressão é um número inteiro ou uma soma de duas expressões e, assim, um programa escrito em `Arit` consiste em uma seqüência de somas de expressões númericas. A partir dos dois programas $\mathcal{SCRIPT}$ que foram apresentados é possível ter uma idéia perfeita acerca do funcionamento e da organização da Linguagem `Arit`.

3.1 Estrutura de Módulos em *SCRIPT*

A estrutura completa de uma definição *SCRIPT* é formada por um módulo principal e um ou mais módulos secundários externos que podem ser compilados junto com o módulo principal ou extraído de uma biblioteca de módulos compilados.

A função básica de um módulo é permitir que entidades relacionadas, tais como domínios e funções, sejam agrupadas para poderem ser usadas por outros módulos. Com isso, certos detalhes de definições de módulos e domínios que não precisam ser conhecidos pelos usuários de um módulo podem ficar ocultos.

Há três tipos de módulos em *SCRIPT*: PROJECT, SYNTAX e MODULE.

3.1.1 Módulo PROJECT

O módulo PROJECT serve para definir os parâmetros e o ambiente sobre o qual as definições devem ser avaliadas.

Na seção de importação deste módulo, somente uma função pode ser importada, a qual deve ser considerada como função principal da definição formal.

Recomenda-se, por razões de clareza, que o domínio da função principal seja redefinido na seção DOMAINS do módulo PROJECT. É necessário que a função principal tenha o mesmo domínio no módulo que a exportou.

As associações entre arquivos e domínios dos argumentos da função principal são definidas nas seções INFILES e OUTFILES. Tais associações servem para estabelecer onde os argumentos de entrada se encontram e onde os resultados devem ser registrados.

A seção COMPONENTS apresenta os arquivos dos módulos com as definições formais.

3.1.2 Módulo SYNTAX

O módulo SYNTAX normalmente apresenta três seções:

DOMAINS: especifica os domínios de símbolos não-terminais.

LEXIS: declara as estruturas dos símbolos léxicos.

SYNTAX: define as sintaxes concretas e abstratas.

3.1.3 Módulo MODULE

Os módulos MODULE permitem encapsular definições de domínios e funções podendo também ser usados para estabelecer a interface de comunicação entre os módulos. Tais módulos são compostos de quatro seções:

seção EXPORTS: apresenta as entidades do módulo corrente que estão sendo exportadas, assim como seus níveis de encapsulamento.

seção IMPORTS: são feitas as importações de símbolos, definindo seus graus de visibilidade e relacionando os módulos de onde eles serão importados.

seção DOMAINS: apresenta as declarações de domínios, variáveis e funções.

seção DEFINITIONS: apresenta as definições de funções e outros valores, por meio de uma lista de definições de funções.

4 O Papel da Linguagem $\mathcal{HASKELL}$ no Projeto

O interpretador que pode ser obtido via uma descrição $\mathcal{SCRIPT}$ é gerado a partir das funções semânticas que aparecem no módulo denominado `MODULE`, presente naquela descrição. Infelizmente, porém, um programa $\mathcal{SCRIPT}$ não pode ser diretamente compilado para linguagem de máquina e executado, pois ainda não existe um compilador deste tipo. Desta forma, a fim de serem executados, os programas escritos em $\mathcal{SCRIPT}$ precisam ser traduzidos para alguma outra linguagem que possua um compilador ou um interpretador operacional, como, por exemplo, as linguagens *C* [12] ou *Java* [7]. A linguagem de programação escolhida como alvo neste processo de tradução chama-se $\mathcal{HASKELL}$ [8]. A principal razão que levou à escolha de $\mathcal{HASKELL}$ foi o fato desta ser uma linguagem pertencente ao paradigma funcional, assim como $\mathcal{SCRIPT}$. Caso fosse definida alguma linguagem imperativa, por exemplo: *Pascal* ou *C*, como alvo no processo de tradução, haveria grande dificuldade, pois estas linguagens são muito diferentes daquelas chamadas funcionais.

Além de muito semelhante a $\mathcal{SCRIPT}$ a linguagem $\mathcal{HASKELL}$ possui algumas vantagens que são típicas de linguagens funcionais, por exemplo:

- Avaliação *Lazy*, isto é, os parâmetros de uma função são avaliados apenas quando necessários no corpo daquela função.
- Polimorfismo universal paramétrico, ou seja, a capacidade de definir funções que se comportam da mesma maneira para parâmetros de tipos diferentes.
- Funções de ordem superior, isto é, funções que podem ser passadas como parâmetros para outras funções, ou retornadas como o resultado de alguma função.
- Funções parciais, ou seja, uma função, quando aplicada a apenas alguns de seus argumentos, retorna uma outra função, mais específica, que pode receber os argumentos restantes da função original.

```
data Exp = Node1 Exp QuotPLUS Exp | Node2 N

data QuotPLUS = Node3

type Q = String
type N = Int
type T = Bool

elab_exp exp =
    case exp of
        Node1 exp1 quotPLUS exp2 ->
            let exp1l = elab_exp exp1 ; exp2l = elab_exp exp2 in exp1l + exp2l
        Node2 n -> n
```

Figura 8: Interpretador de `Arit`, escrito em $\mathcal{HASKELL}$

- Estruturas de dados de tamanho infinito, como por exemplo listas. Esta é uma decorrência direta da avaliação *Lazy*, pois embora tais estruturas não sejam "infinitas" de fato, esta forma de avaliação de parâmetros permite ao programador enxergá-las dessa forma.
- Permite a prova da corretude de programas de forma mais fácil, em grande parte em decorrência da ausência de "efeitos colateriais", isto é, mudanças de estado durante a avaliação de expressões.
- Auxência de variáveis globais e desvios incondicionais, que, quando usados sem disciplina, contribuem para a produção de códigos ilegíveis.

Além dessas características, $\mathcal{HASKELL}$ possui outras que a tornam uma linguagem de programação "limpa" e bem estruturada, adequada às provas de corretude de programas e à geração de código reutilizável. A linguagem ainda dispõe de classes de tipos e funções sobrecarregadas, estas últimas, duas características típicas de linguagens orientadas a objetos.

O texto na Figura 8 é um programa escrito em $\mathcal{HASKELL}$ obtido a partir da tradução do programa $\mathcal{SCRIPT}$ da Figura 7. Este programa foi escrito à mão, mas espera-se que o Gerador de Interpretadores seja capaz de produzir um código semelhante.

Um problema que deve ser mencionado neste relatório diz respeito à forma dos dados de entrada dos interpretadores gerados automaticamente, pois, como já foi dito, um interpretador recebe duas entradas: o código fonte do programa a ser interpretado e os dados de entrada do mesmo. Um programa escrito na linguagem `Arit` se parece com o que pode ser visto na Figura 9, porém para que este programa

```
1 + 2
```

Figura 9: Programa escrito na linguagem `Arit`

```
Node (exp, [ Node (exp, [ Node (1, []) ]), Node (+, []), Node (exp, [ Node (2, []) ]) ])
```

Figura 10: Expressão adequada ao interpretador escrito em $\mathcal{HASKELL}$.

possa ser fornecido como entrada para o interpretador da Figura 8, ele deve ser transformado para o formato visto na Figura 10.

Assim, além de gerar interpretadores, é necessário também que sejam produzidos tradutores capazes de traduzir programas como o da Figura 9 para a forma vista na Figura 10. Esta parte do projeto será abordada na Seção 5.

Nesta seção procurou-se mostrar algumas características interessantes da linguagem $\mathcal{HASKELL}$ e discutir sua importância na implementação de um Gerador de Interpretadores segundo a abordagem deste projeto. Uma descrição mais detalhada desta linguagem funcional pode ser encontrada em [8].

5 Entrada para o Interpretador

Os interpretadores gerados automaticamente não recebem como entrada o código fonte dos programas que devem ser interpretados. Ao invés disso, estes interpretadores recebem uma estrutura de dados que representa a árvore sintática dos programas que serão interpretados. A árvore sintática de um programa mostra como o seu código está estruturado fisicamente. Esta árvore é obtida via as regras de derivação da gramática do programa que está sendo analisado. Cada programa possui uma árvore sintática típica. Esta estrutura pode, então, ser considerada como uma "assinatura" de um programa no contexto de uma linguagem específica. A árvore sintática do programa visto na Figura 9 está mostrada na Figura 11.

Cada linguagem de programação possui uma gramática específica, portanto é necessario que exista um tradutor de código fonte para árvore sintática para cada linguagem, ou seja, neste projeto deve ser produzido não só um Gerador de Interpretadores mas também um Gerador de Tradutores. Esta última ferramenta já foi implementada e maiores informações acerca de seu desenvolvimento podem ser encontradas em [1]. Nesta seção serão vistos apenas alguns aspectos relevantes de seu funcionamento.

O tradutor é obtido a partir do módulo `GRAMMAR` de um programa $\mathcal{SCRIPT}$ que

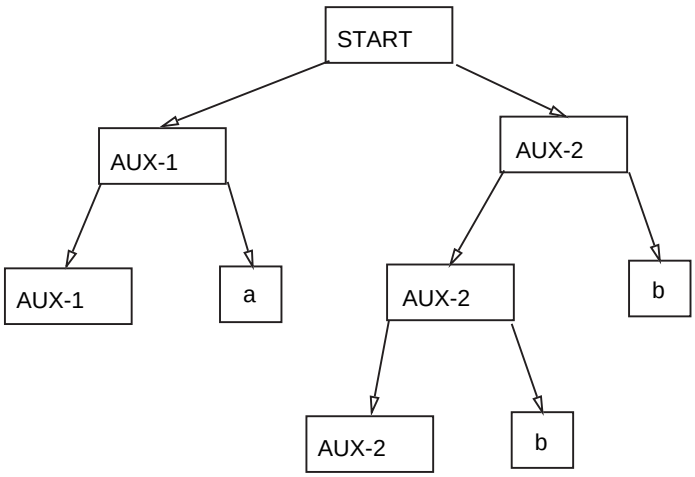

Figura 11: Árvore sintática para o programa "`1 + 2`" escrito na linguagem Arit

descreve uma linguagem $\mathcal{L}$. O Gerador de Tradutores, com base na análise desta parte do programa gera o código de um analisador léxico e de um analisador sintático para $\mathcal{L}$. O código do analisador léxico assim produzido, quando submetido à ferramenta `LEX` resulta em um *scanner* capaz de reconhecer os símbolos de $\mathcal{L}$. O código do analisador sintático, por sua vez, serve de entrada para a ferramenta `YACC`, que a partir dele produz um *parser* capaz de traduzir os programas escritos em $\mathcal{L}$ para o formato adequado ao interpretador. Para maiores informações acerca das ferramentas `LEX` e `YACC` consulte [11]. O esquema apresentado na Figura 12 mostra o processo de geração de tradutores.

A Figura 13 mostra o código do analisador léxico produzido pelo Gerador de Tradutores para o programa $\mathcal{SCRIPT}$ visto na Figura 5. O Gerador de Analisadores Léxicos `LEX`, a partir do programa apresentado na Figura 13 vai gerar um analisador léxico escrito na linguagem `C`. Na Figura 12, `LEX` [11] é um programa capaz de produzir analisadores léxicos escritos na linguagem *C*, `lex.yy.c` é o nome do arquivo gerado por esta ferramenta e lex.l é o nome do arquivo produzido pelo Gerador de Tradutores.

Na Figura 14 pode ser vista uma parte do código do analisador sintático produzido pelo Gerador de Tradutores para a linguagem exemplo `Arit`. Este programa, que será fornecido como entrada para o Gerador de Compiladores `YACC`, contém uma representação da gramática de `Arit`, mais um conjunto de rotinas semânticas que são responsáveis pela tradução de programas escritos em `Arit` para o formato de árvore sintática adequado ao interpretador produzido automaticamente em $\mathcal{HASKELL}$. Na Figura 12, `yacc.y` é o nome do arquivo produzido pelo Gerador de Tradutores. `YACC` [11] é um programa capaz de gerar analisadores sintáticos

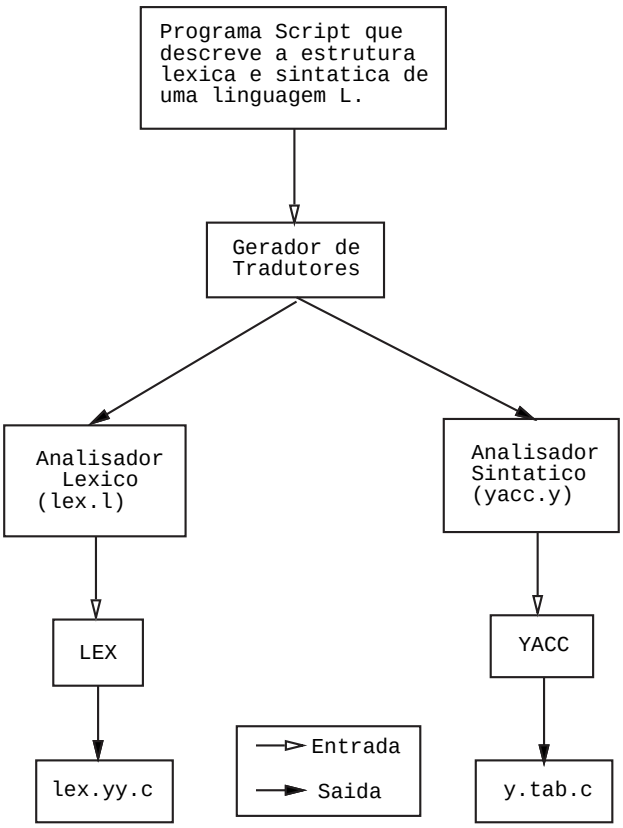

Figura 12: Processo de geração automática de um tradutor

```
%%

"+"             { yylval.quote = strdup(yytext); return(+); }
{digit}+        { yylval.quote = strdup(yytext); return (NUMERO); }
\n              { lineno++; }
.               ;

%%
```

Figura 13: Código de Analisador léxico gerado para `Arit`: `lex.l`

```
%%
aux_1   : exp
        {
          gera_saida ($1);
        }
        ;
exp     : exp '+' exp
        {
          $$ = gera_arv($1, cria_folha(dom($2)));
          $$ = gera_arv($$, $3);
        }
        | id
        {
          $$ = gera_arv($1, NULL);
        }
        ;
id      : NUMERO
        {
          $$ = gera_arv(cria_folha(dom($1)), NULL);
        }
        ;
%%
```

Figura 14: Código do analisador sintático gerado para `Arit`: `yacc.y`

escritos na linguagem *C* e `y.tab.c` é o nome do arquivo produzido por esta ferramenta.

6 Compilador de $\mathcal{SCRIPT}$ para $\mathcal{LAMB}$

Conforme foi dito na Seção 1, este trabalho está inserido no escopo de um projeto maior relacionado à Linguagem $\mathcal{SCRIPT}$. Esta seção vai abordar um outro projeto que também faz parte do escopo de $\mathcal{SCRIPT}$: o Compilador de $\mathcal{SCRIPT}$ para $\mathcal{LAMB}$. Pretende-se, aqui, tratar apenas alguns aspectos dessa ferramenta e, para o leitor mais interessado, recomenda-se a leitura de [3].

A discussão deste compilador é importante porque, estando pronto o Gerador de Intepretadores, pretende-se que sejam integradas as duas ferramentas, por meio da inclusão do código do Gerador ao código do Compilador. Como o Gerador de Tradutores mostrado na Seção 5 já se encontra integrado ao compilador que será visto nesta seção, espera-se que ao final do projeto seja produzida uma única ferramenta com amplas funcionalidades.

6.1 Visão Geral do Compilador

O projeto do compilador $\mathcal{SCRIPT}$ foi estruturado em três passos, como ilustrado na Figura 15. A organização em três etapas se fez necessária, uma vez que a aleatoriedade de programas escritos em $\mathcal{SCRIPT}$ pode fazer com que informações necessárias em um determinado momento ainda não estejam disponíveis, principalmente nos casos de rotinas semânticas recursivas e de declarações que aparecem após o uso. Cada um desses passos é executado por um programa diferente, sendo que o arquivo $\mathcal{SCRIPT}$ que é fornecido como entrada aos compiladores é lido uma vez em cada passo, perfazendo um total de três leituras. Os três compiladores se comunicam por meio de uma tabela de símbolos, que é enriquecida em cada uma das etapas do processo. A seguir, é apresentada uma descrição sucinta dos passos.

Primero Passo As principais tarefas da primeira etapa são:

1. processamento do texto fonte de um programa $\mathcal{SCRIPT}$;
2. análise léxica;
3. análise sintática;
4. coleta e instalação na tabela de símbolos de domínios e variáveis.

Segundo Passo A etapa de análise de dependência e recursividade entre funções e variáveis consiste na criação de listas para indicar quais identificadores aparecem livres em quais definições ou funções. A coleta deste tipo de informação termina com a incorporação das mesmas aos símbolos instalados na tabela de símbolos.

Terceiro Passo Esta etapa consiste na verificação de tipos, análise semântica e geração de código $\mathcal{LAMB}$.

6.2 Integração entre o Compilador e o Gerador

O código do Gerador de Tradutores foi inserido na primeira parte do Compilador de $\mathcal{SCRIPT}$ para $\mathcal{LAMB}$. O Gerador de Tradutores não depende em nada das informações levantadas pelo Compilador, de maneira que, mesmo que o arquivo fonte $\mathcal{SCRIPT}$ contenha erros de tipos, um tradutor será gerado para ele. Isto tem levado a cogitar-se a hipótese de transferir-se todo o mecanismo de geração para a terceira fase do compilador, quando então a verificação de tipos já estaria concluída. Dado que esta mudança representa um trabalho excessivo, será deixada como uma operação opcional, a ser realizada ao final do projeto, se possível.

O Gerador de Interpretadores, ao contrário, deverá ser inserido no compilador responsável pelo terceiro passo do processo de compilação. Como é desejável que

Tarefa	Início	Término
Estudo de Compiladores	19/03/01	07/05/01
Estudo de Haskell	08/05/01	28/05/01
Estudo de Semântica	29/05/01	18/06/01
Estudo de $\mathcal{SCRIPT}$	14/05/01	07/06/01
Projeto do Gerador	08/06/01	29/06/01
Elaboração do relatório parcial	18/06/01	27/06/01

Tabela 1: Cronograma de execução do projeto.

ele obtenha informações da tabela de símbolos construída pelos compiladores, esta parece ser a opção mais acertada. O sistema final deverá ter uma configuração semelhante a que pode ser vista no esquema da Figura 16.

Na Figura 16 P_L representa um programa escrito na linguagem $\mathcal{L}$, que é descrita por um programa $\mathcal{SCRIPT}$. A partir do programa $\mathcal{SCRIPT}$ que descreve $\mathcal{L}$ o Gerador de Tradutores produz um *parser* e um *scanner*, capazes de traduzir programas escritos em $\mathcal{L}$, neste exemplo P_L, para uma estrutura de dados que representa a árvore sintática de P_L. Na terceira fase do processo, o sistema produz um interpretador de $\mathcal{L}$ escrito em $\mathcal{HASKELL}$. Neste exemplo, tal aplicativo foi denominado `interpretador.hs`. Este interpretador é capaz de simular a execução de P_L quando alimentado com a entrada de dados deste programa e com a árvore de sintaxe que o representa.

7 Atividades Desenvolvidas

Buscando cumprir o cronograma previamente estabelecido na proposta de projeto foram desenvolvidas diversas atividades no decorrer do primeiro semestre de 2001. Estas atividades compreendem desde estudos dirigidos até a implementação de código. A Tabela 1 apresenta o cronograma a princípio proposto.

Inicialmente foi realizada uma breve revisão de conceitos relacionados a compiladores, pois uma grande parte da implementação do Gerador de Interpretadores e do Gerador de Tradutores lança mão de técnicas abordadas naquela disciplina. A implementação desses dois geradores se baseia na análise léxica e sintática de programas $\mathcal{SCRIPT}$ e exige conhecimento sobre construção de *scanners*, *parsers* e geração de código [6].

A Linguagem $\mathcal{HASKELL}$ foi o próximo tópico abordado, uma vez que seria utilizada para a codificação dos interpretadores gerados. Foram desenvolvidos diversos programas escritos em $\mathcal{HASKELL}$ durante o aprendizado e conceitos ligados às Linguagens do Paradigma Funcional puderam ser solidificados [8].

Na seqüência, foram estudadas as técnicas de Semântica Denotacional. Esta disciplina busca prover uma notação não ambígua para descrever o comportamento de linguagens de programação. A linguagem $\mathcal{SCRIPT}$ usa este formalismo para descrever o significado de cada comando e expressão de outras linguagens. Maiores referências sobre o assunto podem ser encontradas em [5].

O último tema de estudos foi a Linguagem $\mathcal{SCRIPT}$. Esta ordem foi escolhida porque a análise de $\mathcal{SCRIPT}$ exige conhecimentos relacionados a Linguagens Funcionais e a Semântica Denotacional. Nesta fase foram definidas as semelhanças entre contruções de $\mathcal{HASKELL}$ e de $\mathcal{SCRIPT}$e foi elaborado um esquema de tradução entre as duas linguagens. O estudo desta linguagem foi complementado pela análise do código de um compilador de $\mathcal{SCRIPT}$ para $\mathcal{LAMB}$ [3].

Esta primeira parte do projeto culminou, por fim, com o projeto do Gerador de Interpretadores baseado em $\mathcal{SCRIPT}$. Um pequeno protótipo foi implementado, a fim de testar a viabilidade do sistema. Foram construídas diversas funções para manipular as estruturas de dados que fazem parte do compilador de $\mathcal{SCRIPT}$ para $\mathcal{LAMB}$ descrito em [3].

Além dos estudos dirigidos e do projeto do Gerador de Interpretadores, no primeiro semestre de 2001 também se deu a conclusão do Gerador de Tradutores. Esta ferramenta, descrita na Seção 5, embora ainda sujeita a futuras modificações, encontra-se totalmente operacional.

8 Trabalhos Futuros

Na segunda parte do Projeto Orientado à Computação (POC II) espera-se que o Gerador de Interpretadores baseado em $\mathcal{SCRIPT}$ possa ser implementado. Esta seção descreve algumas questões que devem ser tratadas no decorrer da segunda etapa dos trabalhos. Dois problemas a serem solucionados de imediatos são:

- as diferenças entre algumas estruturas de dados das linguagens $\mathcal{HASKELL}$ e $\mathcal{SCRIPT}$. A definição dos tipos de dados denominados tuplas nas duas linguagens merece destaque especial neste caso;
- a tradução para $\mathcal{HASKELL}$ de uma chamada (em $\mathcal{SCRIPT}$) que pode realizar associação dinâmica.

8.1 Diferenças Conceituais entre $\mathcal{SCRIPT}$ e $\mathcal{HASKELL}$

Embora $\mathcal{HASKELL}$ e $\mathcal{SCRIPT}$ sejam duas linguagens pertencentes ao paradigma funcional, elas possuem inúmeras diferenças. As diferenças sintáticas entre as duas linguagens não representam grande problema para o processo de tradução.

Existem, entretanto, algumas diferenças conceituais que requerem uma análise mais elaborada.

Em primeiro lugar, $\mathcal{SCRIPT}$ possui os conceitos de domínios de tipos e de extensão de domínios que estão ausentes em $\mathcal{HASKELL}$. A Equação 1, por exemplo, define A como o domínio de todas as tuplas que possuem no mínimo dois campos inteiros. Dessa forma, qualquer uma das Equações 2, 3 ou 4 é verdadeira na Linguagem $\mathcal{SCRIPT}$.

$$Def\ A = (N, N) \tag{1}$$

$$(1, 2) \in A \tag{2}$$

$$(1, 2, 3) \in A \tag{3}$$

$$(1, 2, "Lillian") \in A \tag{4}$$

Além disso, $\mathcal{SCRIPT}$ permite que sejam estendidos domínios de tuplas. O programa abaixo, por exemplo, define `A` como o domínio das tuplas que possuem dois inteiros e `B` como o domínio das tuplas que, além de possuírem dois campos inteiros, possuem também um campo com uma cadeia de caracteres:

```
DOMAINS
    A = (N, N)
    B = A EXT (Q)
```

Qualquer função que aceite um membro do domínio `A` como um parâmetro de entrada deve aceitar também um membro do domínio `B`, que constitui uma extensão de `A`:

```
DEFINITIONS
    DEF f(a:A) : A = a
...
    f(a) = a
    f(b) = b
```

A linguagem $\mathcal{HASKELL}$ não possui este conceito de tuplas. Em $\mathcal{HASKELL}$ uma tupla representa um conjunto de elementos no qual a ordem em que eles aparecem é importante e nada mais. Desse modo as Equações 3 e 4 não são verdadeiras nesta linguagem. $\mathcal{HASKELL}$ também não possui capacidade de extensão de tuplas, ou de qualquer tipo de dados, como é o caso da linguagem $\mathcal{SCRIPT}$. A tradução de tuplas de $\mathcal{SCRIPT}$ para algum tipo de dados em $\mathcal{HASKELL}$ não é tão trivial com a tradução de construções mais simples.

8.2 Funções Associadas a Domínios

A linguagem $\mathcal{SCRIPT}$ permite que funções sejam associadas a algum domínio de tuplas. O programa abaixo, por exemplo, define a função `push` como associada ao domínio de tuplas denominado `Stk`.

```
DEF Stk.push(elem) : Stk = ...
```

A aplicação de `push` deve sempre estar associada a uma variável ou objeto que pertença ao domínio `Stk`. Uma função que está associada a um domínio de tuplas, por exemplo, `Stk`, pode ser redefinida e associada a qualquer extensão de `Stk`. Neste caso, os domínios das funções em todas as redefinições devem ser equivalentes.

Redefinições de funções formam uma hierarquia que concorda com as relações de extensão dos domínios associados. A redefinição válida é a mais baixa na hierarquia. Note aqui uma grande semelhança com as linguagens orientadas a objetos, como por exemplo *Java* [7]. No programa a seguir, a função `f`, associada inicialmente ao domínio das tuplas `(N, N)` foi redefinida para o domínio das tuplas `(N, N, N)`. É importante que cada ocorrência da função `f` no corpo de um programa $\mathcal{SCRIPT}$ seja associada a implementação correta. Esta capacidade é denominada "Associação Dinâmica" e está presente em quase todas as linguagens que permitem programação orientada a objetos. Observe que a expressão `f^(x)` representa uma chamada à função `f` associada ao domínio `A`.

```
DOMAINS
    B = A EXT (N, N)

DEFINITIONS
    DEF A.f(x) = ...
    DEF B.f(x) = ... f^(x) ...
    DEF ... b.f(x) ...
```

O impecílho à tradução, neste caso, é o fato da Linguagem $\mathcal{HASKELL}$ padrão não possuir associação dinâmica. Isto não constitui uma deficiência da linguagem, pois todos os conflitos entre tipos e a ligação entre chamadas a funções e a implementação dessas funções são resolvidos de maneira estática, durante a compilação. Este fato, gera, contudo, uma dificuldade extra que precisa ser contornada no processo de tradução proposto neste trabalho.

9 Conclusão

Este relatório apresentou a primeira parte de um Projeto Orientado em Computação cujo objetivo primário é desenvolver uma ferramenta capaz de gerar in-

terpretadores para linguagens de programação mediante a descrição destas linguagens via programas $\mathcal{SCRIPT}$. Nas Seções 3 e 4 foram mostradas as linguagens $\mathcal{SCRIPT}$ e $\mathcal{HASKELL}$. Nas Seções 5 e 6 foram apresentadas duas ferramentas relacionadas a este projeto e nas Seções 7 e 8 foram descritas as atividades já desenvolvidas e as atividades que deverão ser realizadas na segunda etapa deste trabalho, respectivamente.

Conforme é possível inferir via a leitura das seções anteriores, as atividades desenvolvidas no decorrer do primeiro semestre de 2001 foram cumpridas de acordo com o cronograma proposto na etapa inicial do projeto e que pode ser visto na Tabela 1.

Os resultados conseguidos até agora mostram-se animadores, em grande parte porque protótipos mais simples dos interpretadores que serão produzidos automaticamente foram implementados e testados. Estes protótipos foram construídos via a linguagem $\mathcal{HASKELL}$, que até o presente momento, tem-se mostrado uma boa escolha como alvo da tradução de $\mathcal{SCRIPT}$.

Espera-se que, ao fim do segundo semestre de 2001, todo o projeto esteja concluído e que a ferramenta produzida: um Gerador de Interpretadores para Linguagens de Programação, esteja completamente operacional e disponível para outras atividades científicas.

Por fim, é importante que seja dito que, no decorrer deste projeto, foi possível ao aluno ter contato com diversas técnicas de projeto e implementação de compiladores e projeto de linguagens funcionais.

Referências

[1] Pereira, Fernando Magno. Bigonha, Roberto da Silva. Bigonha, Mariza. *Compilação de um programa escrito em linguagem qualquer para $\mathcal{HASKELL}$*. Relatório técnico 001/01. Departamento de Ciência da Computação, Universidade Federal de MInas Gerais, Fevereiro de 2001.

[2] Bigonha, Roberto da Silva. *$\mathcal{SCRIPT}$ 2.1, An Object Oriented Language for Denotational Semantics.* Relatório Técnico 0xx/00. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, julho, 2000.

[3] Oliveira, Fabíola Fonseca. *Compilação de uma Linguagem Funcional, Orientada por Objetos, para Definição de Semântica Denotacional.* Tese de Mestrado, DCC/UFMG, 1998.

[4] Taveira, Wendell Figueiredo. *Compilador da Linguagem Funcional Orientada por Objetos $\mathcal{SCRIPT}$ para $\mathcal{HASKELL}$*. Relatório Final de Projeto de Final

de Curso (POCII). Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, fevereiro, 1999.

[5] Gordon, Michael J. C.. *The Denotational Description of Programming Languages* Springer-Verlag, 1979.

[6] Aho, A.V., Sethi, R. and Ullman, J.D. *Compilers Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[7] Deitel, H. M. Deitel, P. J. *JAVA: como programar*, Bookman, Porto Alegre, 2001.

[8] Thompson, Simon. *Haskell - The Craft of Functional Programming.* Addison-Wesley, 1996.

[9] Bigonha, Roberto da Silva. *The Revised Report on the $\mathcal{SCRIPT}$ Language for Denotational Semantics.* Relatório Técnico 016/97. DCC-UFMG, 07/1997.

[10] Oliveira, Fabíola, Bigonha, Roberto S. Bigonha, Mariza A. S., Costa, Marco R., "Implementação da Linguagem Funcional Script", *Anais do IV Congreso Argentino de Ciencias de la Computacion*, 02/10/2000 a 07/10/2000, Ushuaia, Argentina, página: 539, 2000. Recife-Pernambuco, 32 pages, maio/2000.

[11] Johnson, S. *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, N. J., 1978.

[12] Kerningham, B. and Ritchie, D. *The C Programming Language.* Prentice Hall, 1988.

[13] Bigonha, Mariza A. S., Bigonha, Roberto S., Taveira, Wendell F. e Oliveira, Fabíola F. "Compilador da Linguagem Funcional Orientada por Objetos Script para Haskell", *Anais da XXVI Conferencia Latinoamericana de Informatica - CLEI - PANEL2000,* Trabalho aceito e a ser apresentado e publicado nos anais do CLEI - México setembro de 2000.

Fernando Magno Quintão Pereira

Roberto da Silva Bigonha

Mariza A. S. Bigonha

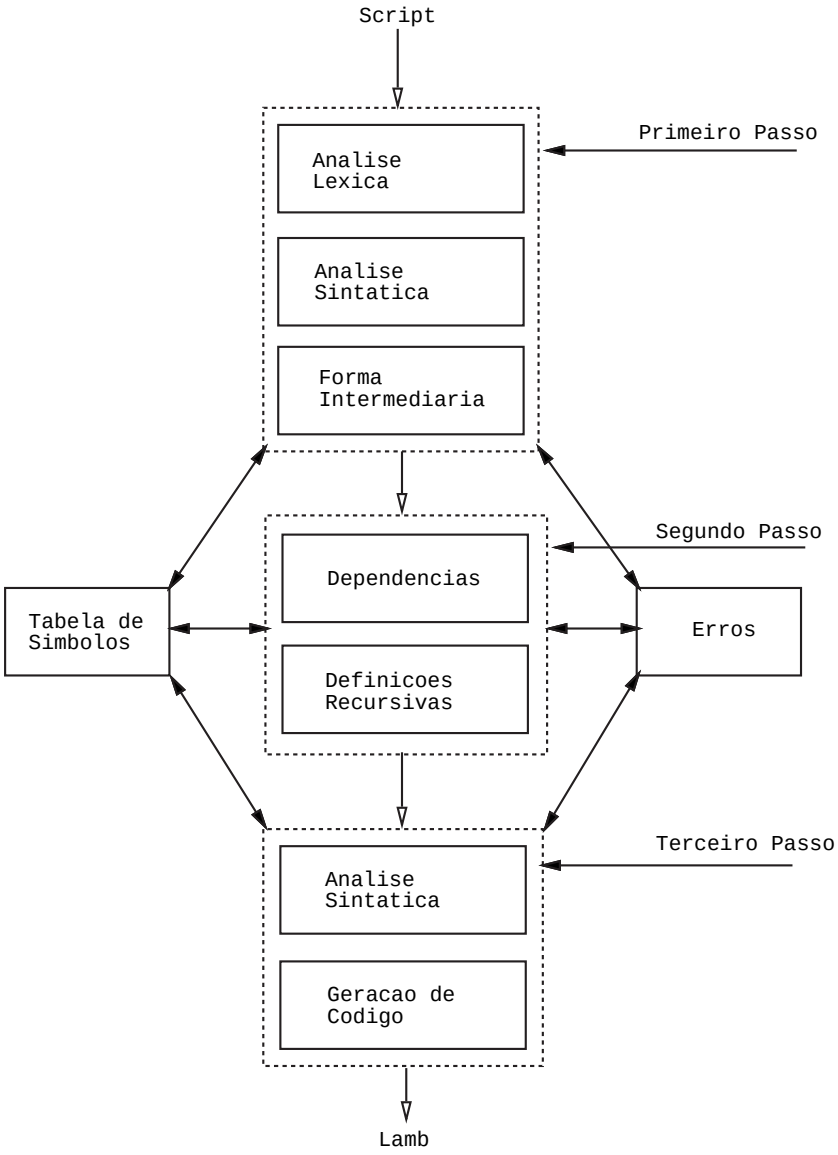

Figura 15: Compilador $\mathcal{SCRIPT}$ para $\mathcal{LAMB}$

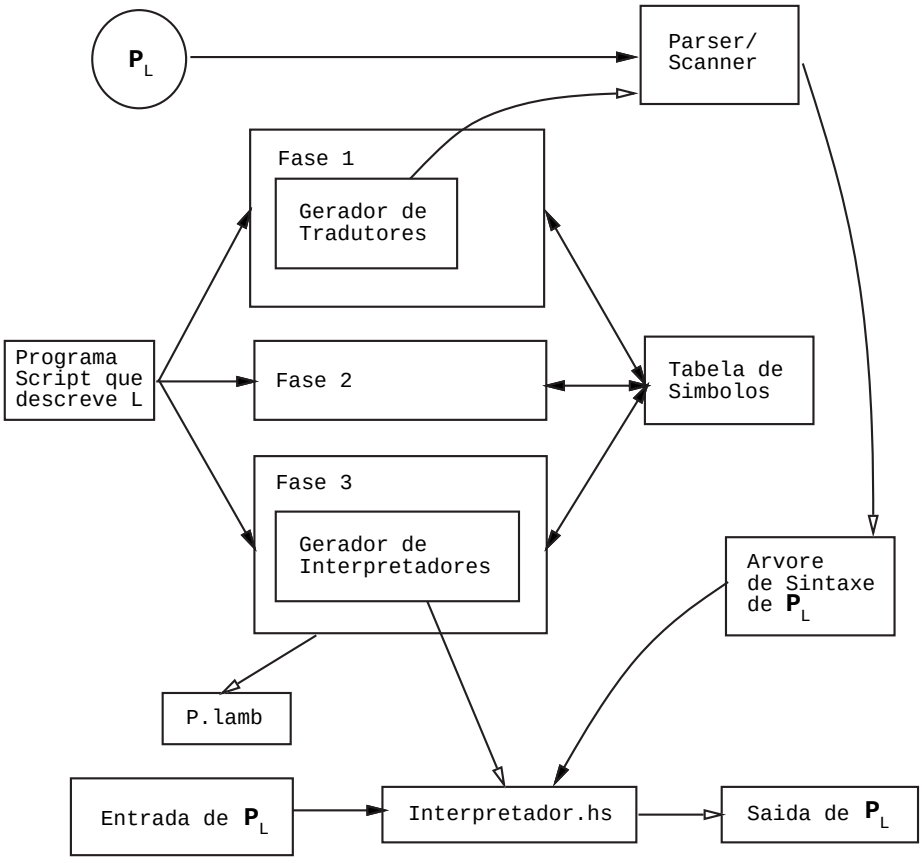

Figura 16: Sistema de compilação e geração de código baseado em $\mathcal{SCRIPT}$