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

Compilação de um Programa escrito em Linguagem de Programação Qualquer para $\mathcal{HASKELL}$

por

Fernando Magno Quintão Pereira

Relatório Técnico do Laboratório de
Linguagens de Programação LLP003/2001

Av. Antônio Carlos, 6627
31270-010 - Belo Horizonte - MG
13 de março de 2001

Sumário

Lista de Figuras

1 Introdução

O trabalho apresentado neste relatório faz parte de um projeto maior que envolve estudantes e professores da área de linguagens de programação da Universidade Federal de Minas Gerais e que se desenvolve em torno de $\mathcal{SCRIPT}$[1], uma linguagem funcional, orientada para objetos, projetada pelo professor Roberto Bigonha. O propósito básico de $\mathcal{SCRIPT}$ é prover uma notação conveniente para descrever o comportamento léxico, sintático e semântico de outra linguagem de programação. Neste documento, uma linguagem qualquer, que pode ser descrita via um programa $\mathcal{SCRIPT}$, será denominada $\mathcal{L}$.

O primeiro trabalho desenvolvido neste contexto tratou-se do projeto e implementação de um compilador de $\mathcal{SCRIPT}$ para $\mathcal{LAMB}$ uma versão extendida do cálculo-λ, a linguagem utilizada como código intermediário na compilação de linguagens funcionais [2]. O segundo trabalho consistiu no desenvolvimento de um compilador de $\mathcal{SCRIPT}$ para a linguagem funcional $\mathcal{HASKELL}$[3]. Além dessas, outras atividades de pesquisa foram desenvolvidas no contexto de $\mathcal{SCRIPT}$: foi feita a tradução de $\mathcal{LAMB}$ para *Supercombinadores* [**?**] e a geração de código, que consistiu na tradução de supercombinadores para código na Linguagem C [9] [**?**].

Neste ponto, o grupo do laboratório de Linguagens de Programação decidiu que era interessante também que fosse criado um método para a geração de interpretadores baseado na linguagem $\mathcal{SCRIPT}$, uma vez que ela fora projetada de forma a permitir que as descrições de semântica denotacional de outras linguagens pudessem ser efetivamente executadas e depuradas. Assim, o que se pretende neste trabalho é, a partir da descrição $\mathcal{SCRIPT}$ de uma linguagem $\mathcal{L}$ qualquer, gerar automaticamente um interpretador para programas escritos em $\mathcal{L}$. Este interpretador será um programa escrito na linguagem funcional $\mathcal{HASKELL}$ que deverá receber a estrutura sintática do programa escrito em $\mathcal{L}$ e a entrada deste programa e produzirá como saída o resultado da execução do programa em $\mathcal{L}$ sobre aquela entrada.

Este documento está organizado da seguinte forma: na Seção 2 serão descritas as etapas necessárias para a geração automática de um interpretador para uma linguagem $\mathcal{L}$ e para a execução de programas escritos em $\mathcal{L}$ via estes interpretadores. Na Seção 3 será discutida a implementação de uma parte deste gerador. Na Seção 4 encontra-se um exemplo de todo o processo de geração de interpretadores para uma linguagem qualquer e, por fim, na Seção 5 serão mostrados os resultados conseguidos até a presente data e a bibliografia consultada.

2 A Geração de Código

O que se pretende neste projeto é implementar um gerador de interpretadores escritos em $\mathcal{HASKELL}$ para uma linguagem $\mathcal{L}$ qualquer a partir de um programa $\mathcal{SCRIPT}$ que define $\mathcal{L}$ formalmente. Este interpretador, contudo, não irá receber o código fonte dos programas escritos em $\mathcal{L}$ diretamante. Antes de serem submetidos, os programas escritos em $\mathcal{L}$ precisam ser traduzidos para uma notação adequada ao interpretador, cuja entrada consiste de uma tupla da linguagem $\mathcal{HASKELL}$ que descreve a estrutura sintática do programa que se pretende interpretar. Assim, além de ser necessário gerar um interpretador para $\mathcal{L}$, é preciso também que seja gerado um tradutor, capaz de traduzir os programas escritos em $\mathcal{L}$ para esta tupla de $\mathcal{HASKELL}$, que será fornecida como entrada ao interpretador.

O processo de geração de um interpretador para $\mathcal{L}$ e de execução de um programa escrito em $\mathcal{L}$ via este interpretador pode ser divido nas três etapas a seguir:

1. A geração do intepretador 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 de $\mathcal{HASKELL}$.

2. A tradução de um programa P_L, escrito em $\mathcal{L}$ para um tupla da linguagem $\mathcal{HASKELL}$, que define a estrutura sintática daquele programa.

3. A submissão da estrutura obtida na fase 2 do processo ao interpretador gerado na fase 1 do mesmo.

Nas próximas subseções serão descritas a linguagem $\mathcal{SCRIPT}$ e cada uma das três etapas necessárias à geração de um interpretador para uma linguagem qualquer e para a execução, neste interpretador, dos programas escritos naquela linguagem.

2.1 Script

Um programa $\mathcal{SCRIPT}$ é formado por três tipos de módulos, cada um deles podendo ser compilado separadamente:

- `PROJECT`: neste módulo é descrito o ambiente no qual os programas $\mathcal{SCRIPT}$ são processados. Entre outras coisas, neste módulo estão inclusas a identificação da função principal da definição formal, os arquivos de entrada e saída de dados associados àquela função, além da identifição dos módulos que compõem toda a definição formal.

- `GRAMMAR`: Este módulo especifica a sintaxe concreta e abstrata de uma linguagem de programação. A estrutura léxica dos símbolos da linguagem é descrita no submódulo `LEXIS`, ao passo que a estrutura sintática da mesma é definida no submódulo `SINTAX`. A partir deste módulo é possível gerar analisadores léxico e sintático capazes de reconhecer a linguagem aqui definida.

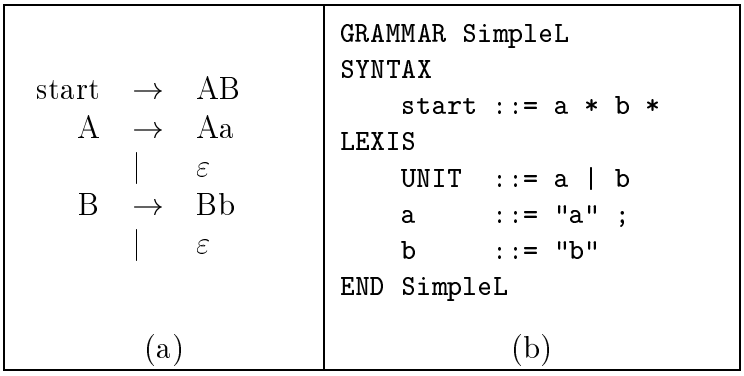
- `MODULE`: Este módulo se presta a encapsular as definições de domínios e de funções para uma linguagem de programação.

Durante o projeto descrito neste documento trabalhou-se apenas sobre o módulo `GRAMMAR` dos programa $\mathcal{SCRIPT}$ uma vez que os elementos sintáticos de uma linguagem de programação, cuja sintaxe é definida neste tipo de módulo permitem que sejam gerados analisadores sintáticos e tradutores para ela, de modo que programas em tal linguagem possam ser convertidos em uma notação de árvore sintática, a partir da qual o comportamento sintático da referida linguagem possa ser inferido.

A fim de ilustrar o que foi dito, a Figura 1 mostra um exemplo de gramática muito simples, que descreve o conjunto de palavras começadas por qualquer quantidade de `a's` e terminada por qualquer quantidade de `b's` e um programa escrito na linguagem $\mathcal{SCRIPT}$, capaz de reconhecer tal gramática.

2.2 Primeira Etapa

Na primeira etapa deste processo é gerado o código de um intepretador para a linguagem $\mathcal{L}$ e o código de um tradutor, capaz de traduzir programas escritos em $\mathcal{L}$ para uma estrutura que representa a árvore de sintaxe abstrata desses programas e que será passada como entrada para o interpretador gerado.



(a)

```
start  → AB
   A   → Aa
       |  ε
   B   → Bb
       |  ε
```

(b)

```
GRAMMAR SimpleL
SYNTAX
    start ::= a * b *
LEXIS
    UNIT  ::= a | b
    a     ::= "a" ;
    b     ::= "b"
END SimpleL
```

Figura 1: (a) Gramática para "a*b*" (b) Programa $\mathcal{SCRIPT}$ para "a*b*".

```
aux-3   : start          {gera_saida($1);}
start   : aux-1 aux-2    {$$ = gera_arv($1, $2);}
aux-2   : aux-2 b        {$$ = gera_arv($1, $2);}
        |                {$$ = gera_arv(NULL, NULL);}
aux-1   : aux-1 a        {$$ = gera_arv($1, $2);}
        |                {$$ = gera_arv(NULL, NULL);}
a       : A-TK           {$$ = gera_arv(cria_folha(dom($1), NULL);}
b       : B-TK           {$$ = gera_arv(cria_folha(dom($1), NULL);}
```

Figura 2: Analisador sintático para a linguagem "a*b*".

O tradutor consiste de um analisador sintático capaz de reconhecer programas escritos em $\mathcal{L}$ e de, paralelamente à análise sintática, traduzi-los para o código de uma árvore sintática abstrata. Além do analisador sintático, nesta primeira fase do processo também é gerado um analisador léxico, que permite o reconhecimento dos símbolos que compõem o alfabeto de $\mathcal{L}$.

O analisador sintático da Figura 2, adaptado para a ferramenta `yacc`, foi obtido a partir do programa $\mathcal{SCRIPT}$ da Figura 1. O analisador da Figura 2, além de reconhecer os programas sintaticamente corretos, escritos em $\mathcal{L}$ gera como sua saída uma representação da árvore abstrata que contém a estrutura sintática daqueles programas.

Além do analisador sintático, é gerado também um analisador léxico para reconhecer os símbolos presentes no alfabeto de $\mathcal{L}$. Este analisador léxico é criado a partir da parte `LEXIS` do módulo `GRAMMAR` de um programa $\mathcal{SCRIPT}$. Infere-se desta observação, então, que o analisador sintático é obtido a partir do bloco `SYNTAX`, que junto ao submódulo `LEXIS`, forma o módulo `GRAMMAR` de um programa $\mathcal{SCRIPT}$. A Figura 3 mostra uma cópia sucinta do "scanner", adaptado para ser usado pela ferramente `lex`.

```
a                { return A-TK; }
b                { return B-TK; }
{any thing else}                ; /* do nothing */
```

Figura 3: Analisador Léxico para "a*b*".

2.3 Segunda Etapa

Após a geração dos analisadores léxico e sintático passa-se à segunda etapa do processo de execução de um programa escrito em $\mathcal{L}$. Como o interpretador gerado na primeira fase do processo não pode receber diretamante o código fonte dos programas escritos em $\mathcal{L}$, é preciso que estes sejam traduzidos para uma forma adequada àquele interpretador, que espera como entrada uma estrutura que representa a árvore de sintaxe abstrata do programa que deverá ser interpretado. Esta segunda fase consiste, então, na tradução do programa que se pretente interpretar para tal estrutura.

A árvore de sintaxe abstrata de um programa P escrito na linguagem $\mathcal{L}$ é obtida por meio da analise léxica e sintática de P feita por meio dos analisadores léxico e sintático de $\mathcal{L}$, gerados na fase anterior. Continuando ainda com a linguagem exemplo mostrada na Figura 1, a árvore sintática abstrata que seria gerada para a entrada `abb` tem a estrutura ilustrada na Figura 4.

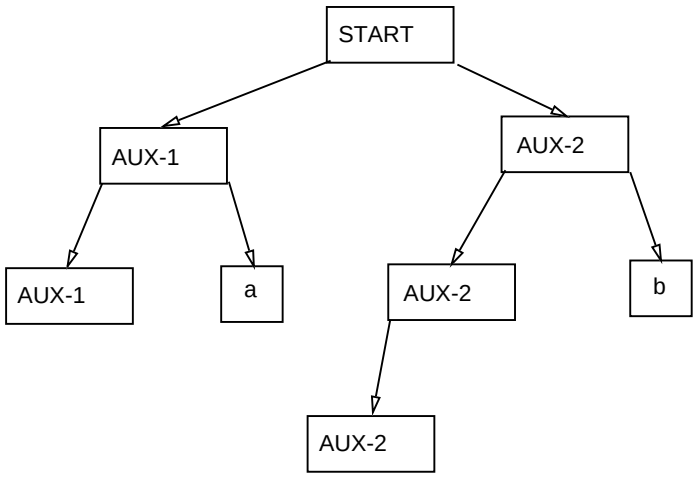


Figura 4: Árvore abstrata gerada para a entrada "abb"

Esta árvore é representada por meio da notação de listas e tuplas utilizadas na linguagem $\mathcal{HASKELL}$ [5]. Cada tupla consiste de dois elementos: um nome de domínio e uma lista contendo os próximos nodos ,tuplas, que são derivadas a partir do elemento cujo domínio foi dado, de forma que, para o exemplo em questão, a saída gerada seria a estrutura:

$$(Start, [(Aux-1, [(Aux-1, []), (A, [])]), (Aux-2, [(Aux-2, [(Aux-2, []), (B, [])]), (B, [])])])$$

Nesta estrutura A é tido como o domínio do símbolo `a` e B é tido como o domínio de `b`. Esta estrutura é ilustrada no formato de árvore binária na Figura 5.

2.4 Terceira Etapa

Após ter sido gerada a árvore de sintaxe abstrata procede-se à terceira etapa do processo, na qual esta árvore, que descreve a estrutura de um programa escrito em $\mathcal{L}$, mais a entrada deste programa, são submetidos ao interpretador escrito em $\mathcal{HASKELL}$. Este interpretador foi produzido junto com o gerador de analisadores sintático e léxico, a partir de um programa $\mathcal{SCRIPT}$ que continha a definição formal da linguagem $\mathcal{L}$. Ao receber a descrição sintática de um programa escrito em $\mathcal{L}$ mais a entrada deste programa, o interpretador retornará como saída o resultado da execução daquele programa sobre aquela entrada.

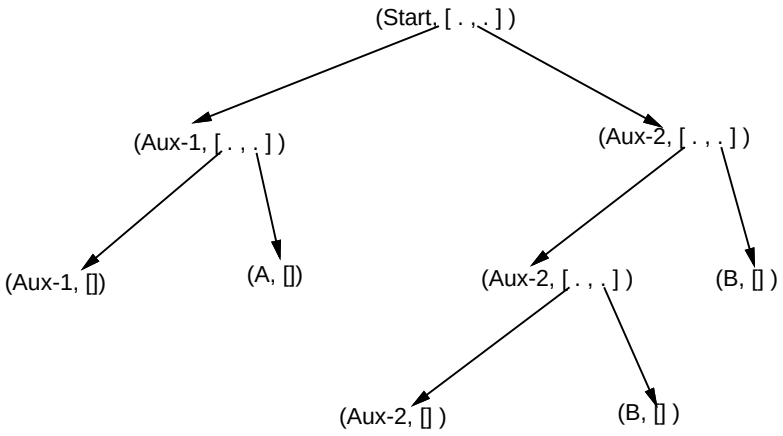


Figura 5: Notação Haskell para árvore abstrata

Todo este processo que foi apresentado em três fases tem por objetivo possibilitar a geração de interpretadores para qualquer linguagem (linguagem $\mathcal{L}$), de forma que quando algum programa P, escrito em $\mathcal{L}$, mais sua entrada I, forem submetidos ao interpretador assim construído, ele seja capaz de retornar a correta saída O que é esperada da execução de P sobre I, de acordo com o esquema presente na Figura 6. É importante que fique claro que o interpretador gerado automaticamente não recebe o código fonte do programa P como uma de suas entradas. Este interpretador recebe a árvore de sintáxe abstrata que descreve aquele programa P, escrita na notação de tuplas e listas da linguagem $\mathcal{HASKELL}$.

Figura 6: Visão geral de um programa em execução.

Todo o processo descrito nesta seção encontra-se resumido nos esquemas das Figuras 7, 8 e 9. Cada um desses esquemas diz respeito a uma das fases de geração de interpretadores escritos em $\mathcal{HASKELL}$ para a linguagem $\mathcal{L}$. Observe que o interpretador para os programas escritos em $\mathcal{L}$ já é gerado logo na primeira fase do processo, sendo utilizados, porém, apenas na última etapa dele.

3 A Implementação

3.1 Visão geral

A implementação do gerador de analisadores léxico e sintático pode ser dividida em dois grandes momentos: no primeiro deles, o gerador foi projetado e implementado de maneira independente do restante dos programas já criados em torno da linguagem $\mathcal{SCRIPT}$. No segundo momento, por outro lado, procedeu-se justamente à inserção do código do gerador, implementado de forma independente, ao compilador de $\mathcal{SCRIPT}$ para $\mathcal{HASKELL}$ criado pelos alunos do Laboratório de Linguagens de Programação: Fabíola Oliveira e Wendell

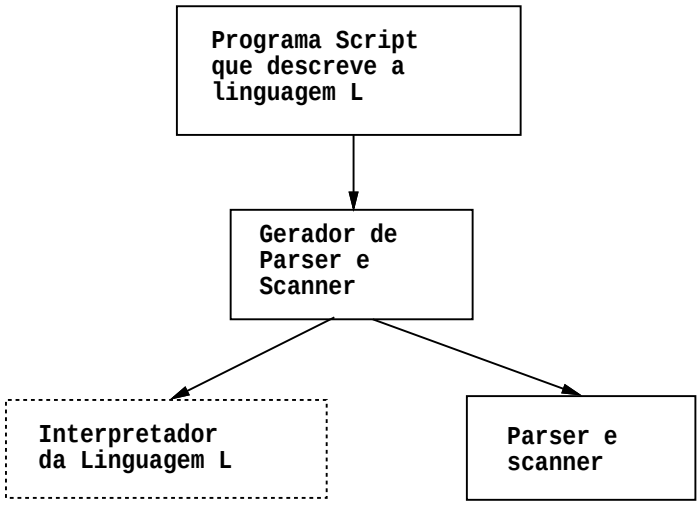


Figura 7: Primeira Fase

Figura 8: Segunda Fase

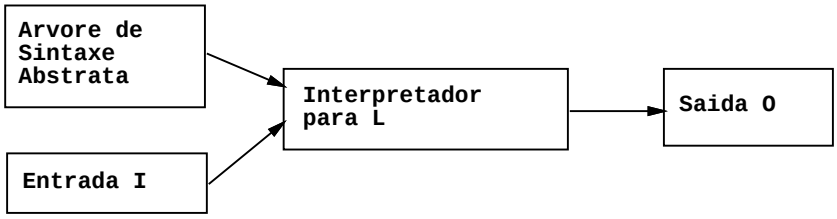


Figura 9: Terceira Fase

Taveira [2, 3].

A implementação do gerador de analisadores léxico e sintático tomou por base a gramática definida no módulo `GRAMMAR` da linguagem $\mathcal{SCRIPT}$. Este módulo se divide em dois submódulos: `SYNTAX` e `LEXIS`, que dizem respeito, respectivamente, à definição sintática de uma linguagem e à descrição léxica dos símbolos que a compõem. O analisador sintático é, dessa forma, gerado a partir da Seção `SYNTAX` do módulo `GRAMMAR` de um programa $\mathcal{SCRIPT}$, ao passo que o analisador léxico é produzido a partir da parte `LEXIS` daquela mesma fonte.

O gerador de analisadores léxico e sintático foi escrito na linguagem C e implementado com o auxílio das ferramentas `yacc` e `lex` [8]. Estas duas ferramentas: `lex` e `yacc` são utilizadas para fazer a análise léxica e sintática do programa $\mathcal{SCRIPT}$ que é passado como entrada para o gerador. Em verdade, todo o gerador de analisadores léxico e sintático foi desenvolvido em torno da gramática de $\mathcal{SCRIPT}$, mais especificamente, em torno do trecho desta gramática que define o módulo `GRAMMAR` de um programa escrito nesta linguagem. O programa `lex` permitiu o reconhecimento dos símbolos que compõem o vocabulário das palavras aceitas por

$\mathcal{SCRIPT}$. A ferramenta `yacc`, por seu turno, possibilitou a análise sintática dos códigos de entrada.

A saída do gerador: código fonte para um analisador sintático e um analisador léxico, é construída paralelamente à análise sintática do programa $\mathcal{SCRIPT}$ de entrada, pelas rotinas semânticas apropriadas. O gerador de analisadores foi implementado com o auxílio das ferramentas `lex` e `yacc` e os próprios programas gerados automaticamante (analisadores léxico e sintático) contituem código fonte a ser passado ao `lex` (analisador léxico) e ao `yacc` (analisador sintático), como será explicado na Subseção 3.4.

3.2 Código fonte

Como já fora dito na Subseção 3.1, este projeto: implementação de um gerador de analisadores sintático e léxico, foi dividido em duas partes: a implementação independente do gerador e a sua posterior incorporação ao compilador de $\mathcal{SCRIPT}$ para $\mathcal{HASKELL}$. Todo o código implementado durante a criação independente do gerador de analisadores léxico e sintático está dividido entre os seguintes arquivos:

syntax.y: este arquivo contém o código fonte do analisador sintático para programas em $\mathcal{SCRIPT}$. Além de permitir o reconhecimento de arquivos de entrada sintaticamente corretos e de informar os possíveis erros de sintaxe, este arquivo contêm as rotinas semânticas que permitem a geração do código dos analisadores sintático e léxico em paralelo à análise sintática da entrada;

syntax.l: este arquivo contêm o código fonte para o analisador léxico encarregado de fornecer ao analisador sintático de $\mathcal{SCRIPT}$ os símbolos de entrada. A comunicação entre este programa e o analisador sintático, cujo código está armazenado no arquivo `syntax.y`, se dá via uma tabela de símbolos;

syntax.h: Este arquivo contêm a maior parte das estruturas de dados necessárias para o armazenamento e a propagação de símbolos, utilizadas durante a análise sintática da entrada e durante a geração automática de código. Entre as estruturas declaradas neste arquivo encontram-se tabelas e listas. Além dessas definições, neste arquivo também constam declarações de variáveis globais e de funções auxiliares implementadas no arquivo `syntax.c`;

syntax.c: neste arquivo estão implementadas grande parte das funções auxiliares necessárias durante a geração automática do código dos analisadores sintático e léxico. Dentre as funções mais importantes implementadas neste arquivo destacam-se aquelas responsáveis por:

- manipular a tabela de símbolos responsável pela comunicação entre o analisador léxico (`syntax.l`) e o analisador sintático (`syntax.y`);
- acessar as listas encadeadas responsavéis pela propagação de símbolos durante a geração automática do código de saída;
- imprimir o código final gerado durante o processo, em arquivos de texto;

slista.h: este arquivo contém a interface para um conjunto de operações utilizadas para manipular uma lista simples, ou seja, encadeada em uma única direção. Além da definição da estrutura desta lista, este arquivo também contém as declarações das funções implementadas no arquivo `slista.c`;

```
GRAMMAR Soneto
SYNTAX
  soneto ::= titulo sep estrofe sep estrofe sep estrofe sep estrofe
             : [titulo estrofe estrofe estrofe estrofe] ;
  sep   ::= "\n\n" ;
  titulo ::= palavra+ ;
  estrofe ::= palavra+
LEXIS
  UNIT  ::= palavra ;
  palavra ::= letra+ : QUOTE letra+ ;
  letra === "a".."z" | "A".."Z"
END Soneto
```

Figura 10: `soneto.spt`: Programa escrito na linguagem $\mathcal{SCRIPT}$.

slista.c: este arquivo contém a implementação das operações definidas para o Tipo Abstrato de Dados Lista Simples. Esta é uma lista encadeada em uma direção apenas e, no gerador de analisadores sintático e lexico, se presta a armazenar simbolos até que sejam necessários de alguma forma. A interface para este programa encontra-se no arquivo `slista.h`;

3.3 Interface com o Usuário

Todo o código foi implementado e compilado em uma máquina SUN, executando o sistema operacional UNIX. Para compilar o código fonte descrito na Subseção 3.4, digite os seguintes comandos:

```
lex syntax.l
yacc -d syntax.y
gcc slista.c syntax.c y.tab.c lex.yy.c -o lds -ll -ly
```

Esta seqüência de comandos irá gerar um arquivo executável denominado `lds`. Para utilizar este arquivo, digite o seu nome, seguido pelo nome de um arquivo que armazene um programa $\mathcal{SCRIPT}$, como, por exemplo, aquele contido na Figura 10. Daqui por diante, neste documento, a extensão `.spt` será utilizada para caracterizar os arquivos contendo programas $\mathcal{SCRIPT}$. Estes arquivos contem programas que descrevem a estrutura léxica, sintática e semântica de alguma linguagem de programação, que neste documento recebem a denominação de $\mathcal{L}$.

A saída gerada pelo comando `lds <arquivo.spt>` será explicada na Seção 3.4 deste documento.

3.4 A Saída Produzida pelo Gerador de Analisadores Léxico e Sintático

Ao executar o comando `lds <arquivo.spt>`, considerando que `<arquivo.spt>` seja um nome de arquivo que contenha um programa $\mathcal{SCRIPT}$, como por exemplo o arquivo `soneto.spt`, cujo conteúdo pode ser visto na Figura 10, serão gerados vários arquivos de saída, a saber:

syntax.make: este arquivo contém as diretivas de compilação para os analisadores sintático e léxico, recém gerados. A fim de compilar o código obtido, ele faz uso da ferramenta `make` do sistema UNIX;

lex.l: este arquivo contém o código do analisador léxico gerado automaticamente. Este programa se presta a reconhecer os símbolos léxico definidos no submódulo `LEXIS` do programa $\mathcal{SCRIPT}$ de entrada;

yacc.y: este arquivo contém o código do analisador sintático gerado automaticamente a partir do submódulo `SYNTAX` do programa $\mathcal{SCRIPT}$ de entrada. Este programa $\mathcal{SCRIPT}$, como já foi mencionado, descreve uma linguagem de programação. O analisador sintático gravado no arquivo `yacc.y` se presta a reconhecer os programas sintaticamente corretos escritos naquela linguagem. Este analisador contém ainda as rotinas semânticas necessárias à geração de uma árvore de sintaxe abstrata que descreve a semântica dos arquivos que lhe são fornecidos como entrada, do modo descrito na Seção 2 deste relatório. Além deste código, este arquivo contém a implementação das funções responsáveis pela manipulação da árvore de sintaxe abstrata;

header.h: este arquivo contém uma interface para as funções que manipulam a árvore de sintaxe abstrata, gerada pelo analisador sintático contido no arquivo `yacc.y`, além da definição dos tipos de dados utilizados na implementação de tal estrutura.

São gerados ainda mais dois arquivos, estes contudo, não são considerados parte da saída padrão, pois se prestam apenas como auxiliares na geração automática de código. São eles:

lex.aux: este arquivo serve com auxiliar durante a geração de `lex.l` , armazenando temporariamente as definições regulares que, posteriormente, farão parte do código final;

gramatica: este arquivo serve como auxiliar durante a geração de `yacc.y`, armazenando a parte final de seu código.

3.5 Integração do Gerador de Analisadores Léxico e Sintático ao Compilador de $\mathcal{SCRIPT}$

O compilador [2, 3] introduzido na Seção 1 deste documento procedia a tradução do módulo `MODULE` de programas $\mathcal{SCRIPT}$ para programas $\mathcal{HASKELL}$. O bloco `GRAMMAR` dos programas $\mathcal{SCRIPT}$ não era tratado por este compilador, sendo necessário, portanto que fosse incorporado ao seu código os procedimentos necessários para que, junto com o interpretador $\mathcal{HASKELL}$ fossem gerados também analisadores léxico e sintático para a linguagem de programação descrita pelo programa $\mathcal{SCRIPT}$ de entrada.

A integração das rotinas semânticas necessárias à geração de analisadores léxico e sintático ao código do compilador foi simples, principalmente por este ter sido projetado de forma bem estruturada e apresentar grande legibilidade. O processo de compilação de um programa $\mathcal{SCRIPT}$ para $\mathcal{HASKELL}$ foi separado em três passos distintos, sendo que cada um desses passos possui seus próprios analisadores léxicos e sintáticos, além de funções auxiliares.

As rotinas semânticas necessárias à geração automática de código foram introduzidas apenas no primeiro dos três analisadores sintáticos. Todos os três analidadores sintáticos, contudo, sofreram pequenas mudanças em suas gramáticas, uma vez que, no decorrer do projeto, à própria gramática de $\mathcal{SCRIPT}$ fora modificada, havendo portanto diferenças sutis entre o que havia sido proposto inicialmente e a versão final implementada. A Figura 11 apresenta uma visão geral de todo o processo de execução de um programa escrito em uma linguagem qualquer $\mathcal{L}$ por meio de um interpretador de $\mathcal{L}$ gerado automaticamente.

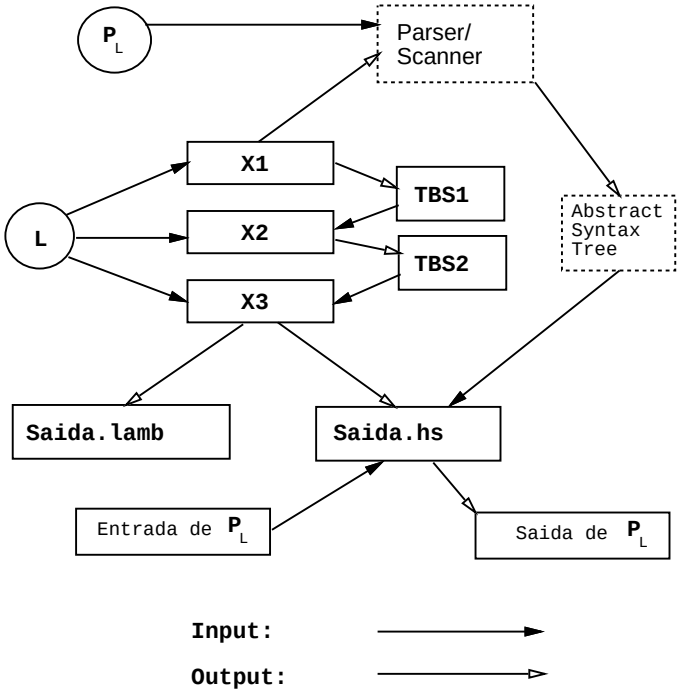


Figura 11: Visão geral

Na Figura 11, cada um dos nomes envoltos em caixas faz referência a um dos arquivos que serão descritos a seguir. Como o processo de compilação de um programa $\mathcal{SCRIPT}$ para $\mathcal{HASKELL}$ foi dividido em três passos, existem três compiladores, os quais foram denominados: `X1, X2, X3`.

X1 é o compilador responsavél pelo primeiro passo. Suas principais tarefas são:

1. processamento do texto fonte;
2. análise léxica;
3. análise sintática;
4. coleta e instalação na tabela de símbolos de domínios e variáveis.

X2 é o compilador responsável pelo segundo passo, correspondente à fase de análise de dependência e recursividade entre funções e variáveis;

X3 é o compilador referente ao terceiro passo do processo, que consiste na verificação de tipos, análise semântica e geração de código;

L é o programa fonte de entrada, escrito na linguagem $\mathcal{SCRIPT}$ e que descreve uma certa linguagem $\mathcal{L}$. O que se pretende é gerar um compilador desta linguagem para $\mathcal{HASKELL}$;

TBS1 é a primeira tabela de símbolos gerada. Esta tabela possui as seguintes aplicações:

1. gerência de escopo e visibilidade dos símbolos;
2. inserção, remoção e consulta de símbolos;
3. exportação e importação de símbolos.

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

Figura 12: Programa $\mathcal{SCRIPT}$ para somas de números inteiros.

TB2 é a segunda tabela de símbolos. A segunda etapa do processo de compilação termina com a incorporação das informações coletadas naquele passo aos símbolos já intalados;

saida.hs contém o código $\mathcal{HASKELL}$ gerado a partir do programa $\mathcal{SCRIPT}$ de entrada. Este funciona como um interpretador para os programas escritos na linguagem $\mathcal{L}$;

saida.lamb contém o código escrito em $\mathcal{LAMB}$, gerado a partir da compilação do programa $\mathcal{SCRIPT}$ de entrada. Este é um resquício dos primeiros dias do projeto, quando o objetivo era traduzir programas $\mathcal{SCRIPT}$ para $\mathcal{LAMB}$, a linguagem intermediária do cálculo λ [2];

parser/scanner são os analisadores léxico e sintático, gerados como resultado das modificões levadas adiante nesta fase do projeto. Estes analisadores recebem como entrada programas escritos na linguagem $\mathcal{L}$ e geram sua árvore de sintaxe abstrata.

O resultado final conseguido nesta fase foi um programa capaz de traduzir arquivos fonte $\mathcal{SCRIPT}$ para $\mathcal{HASKELL}$ e de gerar analisadores léxico e sintático para a linguagem especificada naquele texto de entrada. O processo, contudo, ainda não funciona com perfeição, uma vez que os analisadores gerados automaticamente necessitam de pequenas melhorias, as quais serão discutidas na Seção 5.

4 Um Exemplo

Nesta seção será apresentado um exemplo de saída produzida pelos analisadores léxico e sintático, gerados automaticamante a partir de um programa $\mathcal{SCRIPT}$muito simples que contem a descrição sintática e léxica de uma linguagem capaz de reconhecer somas de números inteiros. Este programa pode ser visto na Figura 12.

A partir desta definição de linguagem, escrita em $\mathcal{SCRIPT}$ será gerado um analisador léxico, capaz de reconhecer os números inteiros e um analisador sintático, capaz de reconhecer expressões formadas por somas aritméticas desprovidas de parênteses. Nas próximas páginas podem ser vistos cada um dos quatro arquivos gerados automaticamente a partir do programa mostrado na Figura 12.

Este é um dos arquivos criado pelo gerador de analisadores léxico e sintático a partir do programa $\mathcal{SCRIPT}$ mostrado na Figura 12. Este arquivo descreve as estruturas de dados utilizadas na construção da árvore abstrata:

```
#ifndef _teste_h
#define _teste_h

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef enum
{
  interno, externo
} nodo_tipo;

typedef struct nodo_str *arvore;

typedef struct nodo_str
{
  nodo_tipo nt;
  union
  {
    struct
    {
      arvore esq, dir;
      char *label;
    } noh;
      char *chave;
  } broto;
} nodo;

arvore gera_arv (arvore a1, arvore a2, char *label);
arvore cria_folha (char *s);
char *dom (char *s);

#endif
```

Durante a construção da árvore de sintaxe abstrata usa-se de fato uma estrutura de árvore binária, com nodos de dois tipos: nodos internos, para gravar os nomes de símbolos não-terminais da gramática e nodos externos, para armazenar os nomes dos símbolos terminais. Um nodo interno pode guardar o nome do símbolo não terminal que ele está representando. Os nodos internos que não contêm estes rótulos são considerados extensões de outros nodos, de forma que é possível representar os vários descendentes de um único símbolo não-terminal.

Este é o código do analisador léxico escrito para servir de entrada à ferramenta `lex`. Este programa é capaz de reconhecer números inteiros positivos, formados por qualquer dos dígitos decimais, em qualquer ordem, além do sinal de soma: "+":

```
/*
 * Analisador lexico gerado automaticamente.
 */

%{
#include "header.h"
#include "y.tab.h"
int lineno = 1;
%}
/* Regular Definitions */
digit ([0-9])

%%

"+" { yylval.quote = strdup(yytext); return(TOKEN_1); }
{digit}+ { yylval.quote = strdup(yytext); return ID_TK; }
\n { lineno++; }
. ;

%%

yyerror(char *s)
{
fprintf(stderr, "%d: %s %s\n", lineno, s, yytext);
}
```

Este é o código gerado para o analisador sintático. Este arquivo é utilizado como entrada para a ferramenta `yacc`, que a partir dele gera um programa escrito na linguagem C capaz de reconhecer entradas sintaticamente corretas escritas na linguagem da Figura 12. Além das regras gramaticais capazes de reconhecer simples somas de expressões, este programa também contém as rotinas semânticas responsáveis pela geração da árvore de sintaxe abstrata de um dado programa de entrada:

```
/*
 * Analisador syntatico gerado automaticamente
 */

%{
#include "header.h"
%}

%union {
  char *quote;
  int number;
  arvore arv;
}
```

```
%token <quote> TOKEN_1
%token <quote> ID_TK
%type <arv> exp
%type <arv> id

%%
aux_1 : exp
{
  gera_saida ($1);
}
;
exp : exp TOKEN_1 exp
{
  $$ = gera_arv($1, cria_folha(dom($2)));
  $$ = gera_arv($$, $3);
}
| id
{
  $$ = gera_arv($1, NULL);
}
;
id : ID_TK
{
  $$ = gera_arv(cria_folha(dom($1)), NULL);
}
;

%%

int main(int argc, char **argv)
{
  extern FILE *yyin, *yyout;
  if (argc > 3)
  {
    fprintf (stderr, "Sintaxe: %s infile outfile\n", argv[0]);
    return 0;
  }
  if (argc > 1)
    if (!(yyin = fopen(argv[1], "r")))
    {
      fprintf (stderr, "Erro na abertura de %s\n", argv[1]);
      return 0;
    }
  if (argc > 2)
    if ( !(yyout = fopen (argv[2], "w")))
    {
      fprintf (stderr, "Erro na abertura de %s\n", argv[2]);
      return 0;
    }
  else
```

```
    yyout;
  yyparse();
  return 1;
}

char *dom(char *s)
{
  return s;
}

arvore gera_arv (arvore a1, arvore a2)
{
  if (!a1)
    if (!a2)
      return NULL;
    else
      return a2;
  else if (!a2)
    return a1;
  else
  {
    arvore galho = (arvore) malloc (sizeof(nodo));
    galho->nt = interno;
    galho->broto.noh.esq = a1;
    galho->broto.noh.dir = a2;
    return galho;
  }
}

arvore cria_folha (char *s)
{
  arvore folha = (arvore) malloc (sizeof (nodo));

  folha->nt = externo;
  folha->broto.chave = strdup(s);
  return folha;
}

void percorre_arvore (arvore a, char modo)
{
  if (a->nt == interno && a->broto.noh.esq)
    percorre_arvore (a->broto.noh.esq, modo);
  if (a->nt == externo)
    if (modo)
      printf ("%s", a->broto.chave);
    else
      printf ("dom(\"%s\")", a->broto.chave);
  if (a->nt == interno && a->broto.noh.dir)
  {
    printf(", ");
    percorre_arvore (a->broto.noh.dir, modo);
```

```
  }
}

void gera_saida (arvore a)
{
  printf ("( [");
  percorre_arvore (a, 0);
  printf ("], [");
  percorre_arvore (a, 1);
  printf ("] )\n");
}
```

O último dos programas gerados automaticamente contém as diretivas de compilação de todo o código criado. Este arquivo faz uso da ferramenta `make` presente nos sistemas UNIX de modo a prover uma maneira cômoda de compilar todos os arquivos produzidos pelo gerador. Para executar este aplicativo, basta digitar a seguinte sequência de comandos: `make -f syntax.make`. Além de compilar todo o código, este programa também possibilita:

- apagar todos os arquivos objetos gerados durante a compilação: `make -f syntax.make cleanobj`;
- apagar os arquivos gerados pelas ferramentas `lex` e `yacc` durante a compilação (`lex.yy.c` e `y.tab.c`): `make -f syntax.make cleany`;
- apagar todos os arquivos gerados durante a compilação (objetos, `lex.yy.c` e `y.tab.c`): `make -f syntax.make cleanall`.

```
#Diretivas de compilacao.

CC = cc
LEX = lex
YACC = yacc

all: lds

cleanobj:
$(RM) *.o core

cleany:
$(RM) y.tab.* lex.yy.c

cleanall:
$(RM) *.o core y.tab.* lex.yy.c

lds: y.tab.o lex.yy.o
$(CC) -o lds y.tab.o lex.yy.o -ly -ll

lex.yy.o: lex.yy.c "y.tab.h

lex.yy.o y.tab.o: header.h
```

```
y.tab.c "y.tab.h: yacc.y
$(YACC) -d yacc.y

lex.yy.c: lex.l
$(LEX) lex.l
```

5 Conclusão e Trabalhos Futuros

Nesta seção são apresentados os resultados conseguidos até a presente data: fevereiro de 2001, bem como o que ainda há por ser feito de maneira que se consiga criar um gerador de interpretadores capaz de simular a execução de um programa escrito em uma linguagem $\mathcal{L}$ qualquer.

O gerador de analisadores sintático e léxico encontra-se operacional e completamente incorporado ao compilador de $\mathcal{SCRIPT}$ para $\mathcal{HASKELL}$ [2, 3], conforme descrito na Seção 3, deste manual. Este programa tem se comportado bem nos testes realizados até aqui e um exemplo de seu funcionamento pôde ser visto na Seção 4.

O projeto, entretanto, está ainda longe do término, uma vez que resta implementar o gerador automático de interpretadores para linguagens descritas via programas $\mathcal{SCRIPT}$. Estes interpretadores, que serão escritos na linguagem funcional $\mathcal{HASKELL}$, ainda não são gerados automaticamante. A implementação deste gerador de código será, sem dúvida, o principal objetivo dos próximos trabalhos relacionados à linguagem $\mathcal{SCRIPT}$.

Referências

[1] 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.

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

[3] 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.

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

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

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

[7] 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.

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

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

[10] 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.