Relatório do Trabalho de Construção de Compiladores

Gustavo Zacarias Souza - 22.1.4112
Caio Lucas Pereira da Silva - 22.1.4006

25 de janeiro de 2025

Sumário

1 Introdução **2**

2 Decisões de Projeto **2**
2.1 Definição dos Tokens 2
2.2 Estratégia do Analisador Léxico 3
2.3 Estratégia do Parser 3

3 Desenvolvimento **4**
3.1 Estruturas de Dados 4
3.2 Exemplo de Análise 4

4 Conclusão **5**

1 Introdução

Este relatório descreve o processo de desenvolvimento de um compilador para uma linguagem de programação simplificada. O trabalho foi realizado como parte da disciplina de Construção de Compiladores e aborda a implementação de um analisador léxico, um analisador sintático e a interpretação de programas.

Serão apresentados detalhes sobre as decisões de projeto, como o uso de expressões regulares para definição de tokens, a escolha de estruturas de dados, e as estratégias adotadas para a implementação do analisador léxico, do parser e do interpretador. O objetivo principal é facilitar o entendimento das soluções implementadas, bem como fornecer informações para compilar e executar o código entregue.

2 Decisões de Projeto

2.1 Definição dos Tokens

Para a análise léxica, definimos os tokens utilizando expressões regulares. Estas foram elaboradas para reconhecer palavras-chave, operadores, identificadores, números e símbolos da linguagem. Abaixo estão exemplos atualizados das definições de tokens relevantes:

```
$digit = 0-9
$alpha = [a-zA-Z]
@identifier = $alpha[$alpha $digit]*
@number = $digit+

<0> "int" { simpleToken TTInt }
<0> "bool" { simpleToken TTBool }
<0> "float" { simpleToken TFloat }
<0> "char" { simpleToken TChar }
```

Listing 1: Definição de tokens em Alex

Cada token identificado é associado a um tipo no Haskell, conforme o exemplo abaixo:

```
data Lexeme =
    | TIdent String
    | TNumber Int
    | TAssign
    | TPlus
    | TTInt
    | TTBool
```

```
8       | ...
```

Listing 2: Exemplo de associação de tokens a tipos

Essas definições permitem um reconhecimento preciso de cada elemento da linguagem, suportando a extensibilidade para novos tipos de dados e operadores.

2.2 Estratégia do Analisador Léxico

O analisador léxico foi implementado utilizando a ferramenta Alex, que transforma as expressões regulares em um scanner de tokens. A estrutura de estado foi usada para lidar com comentários aninhados e outros casos específicos:

```
1 nestComment :: AlexAction Token
2 nestComment input len = do
3   modify $ \s -> s { nestLevel = nestLevel s + 1 }
4   skip input len
```

Listing 3: Gerenciamento de estados para comentários

2.3 Estratégia do Parser

O analisador sintático foi construído com combinadores recursivos. Isso simplifica a modularidade do código e permite lidar com diferentes expressões e declarações. Segue um exemplo:

```
1 varDeclParser :: Parser Char Decl
2 varDeclParser
3   = Decl <$> varParser <*> (doubleColon *> tyParser)
```

Listing 4: Parser para declarações de variáveis

O parser implementa suporte para funções, estruturas de controle, operadores lógicos e aritméticos, e blocos de código. Entretanto, apesar dos esforços, não conseguimos alcançar a funcionalidade completa esperada. Problemas como ambiguidade em algumas gramáticas e dificuldade em gerenciar certos tipos de erros de entrada comprometeram o desempenho ideal do parser. Ainda assim, acreditamos que a estrutura modular servirá como base para futuras melhorias.

Um exemplo de problema ocorre no suporte para expressões aninhadas e funções com múltiplos retornos, onde o parser não trata corretamente casos mais complexos. Testamos várias abordagens, mas o parser continua apresentando limitações em cenários específicos.

3 Desenvolvimento

Durante o desenvolvimento, enfrentamos desafios como a implementação de comentários aninhados e a criação de um parser que lida com múltiplas operações em cadeia. Apesar disso, avançamos significativamente na construção de componentes como o analisador sintático e léxico.

3.1 Estruturas de Dados

A estrutura de dados principal do programa é definida no módulo `Syntax.hs`. Os componentes chave incluem:

```
data Exp
    = Exp :+: Exp
    | Exp :-: Exp
    | Exp :*: Exp
    | Exp :&&: Exp
    | ENot Exp
    | EValue Value
    deriving (Eq, Show)

data Decl = Decl Var Type
    deriving (Eq, Show)
```

Listing 5: Definição de Expressões e Declarações

Essa abordagem organiza bem os diferentes elementos da linguagem, separando expressões, declarações e valores de forma clara.

3.2 Exemplo de Análise

Para ilustrar o fluxo do compilador, considere o seguinte exemplo de entrada e seu processo:

```
int main() {
  print(10);
}
```

Listing 6: Exemplo de código de entrada

A análise léxica gera os seguintes tokens:

```
[Token TIdent "main", Token TLParen, Token TRParen, ...]
```

O parser então constrói a seguinte estrutura sintática:

```
Program [Function "main" [] TInt [] [Print (EValue (EInt 10))
    ]]
```

No entanto, como mencionado anteriormente, o parser falha em lidar com entradas mais complexas, como funções aninhadas ou expressões compostas, devido a limitações na implementação atual.

4 Conclusão

A implementação do compilador proporcionou uma visão aprofundada das técnicas de análise léxica e sintática, bem como da estruturação de linguagens. Apesar das dificuldades, o projeto demonstrou ser um exercício valioso em modularidade e design de software. O parser, embora funcional em casos simples, não alcançou 100% de eficiência, e essa limitação será um ponto de aprendizado para trabalhos futuros.

Passos para Compilar e Executar

Para compilar e executar o código deste projeto, siga as etapas abaixo:

1. Compile o programa utilizando o `stack`:

```
stack build
```

2. Execute o analisador léxico para gerar tokens:

```
./Lang --lexer nome_do_arquivo
```