Relatório de Projeto: Compilador SL

Rafael Diniz
Matrícula: rafael.do@aluno.ufop.edu.br
BCC328 - Construção de Compiladores I - DECOM/UFOP
20 de Fevereiro de 2026

Resumo

Apenas o TP1 foi implementado, com as funções –lexer e –parser. O analizador léxico não reconhece forall, então apenas os testes de 1 a 5 devem funcionar.

Este relatório apresenta o desenvolvimento de um compilador para a linguagem SL (Simple Language) implementado em Haskell. O projeto inclui a análise léxica utilizando Alex, definição de estruturas sintáticas, sistema de tipos básicos e utilitários para manipulação de valores. O compilador está sendo desenvolvido de forma modular, seguindo boas práticas de engenharia de software e utilizando ferramentas padrão do ecossistema Haskell.

Sumário

1 Introdução

O projeto SL (Simple Language) consiste na implementação de um compilador para uma linguagem de programação simples, desenvolvido como trabalho prático da disciplina BCC328 - Construção de Compiladores I. A linguagem SL suporta tipos básicos (inteiros, booleanos, strings e ponto flutuante), declarações de variáveis, estruturas de controle condicionais e iterativas, além de operações aritméticas e lógicas.

O compilador foi implementado em Haskell, aproveitando as vantagens da programação funcional para construção de compiladores, incluindo casamento de padrões, tipos algébricos e sistema de tipos robusto. O projeto utiliza ferramentas padrão do ecossistema Haskell, como Alex para geração do analisador léxico e Cabal para gerenciamento de dependências.

1.1 Estrutura sintática de SL

A linguagem SL foi projetada com uma sintaxe simples mas expressiva. A gramática inclui:

- **Tipos básicos**: `int`, `float`, `bool`, `string`
- **Declarações**: declarações de variáveis com palavra-chave `let`
- **Expressões**: aritméticas, lógicas, relacionais e de atribuição
- **Comandos**: atribuição, leitura, impressão, condicionais e iteração

A árvore de sintaxe abstrata foi definida no módulo `Sl.Frontend.Syntax.SlSyntax` utilizando tipos algébricos de Haskell, proporcionando representação natural e type-safe das construções da linguagem.

1.2 Sistema de tipos para SL

O sistema de tipos implementado suporta:

- Tipos primitivos: inteiros, booleanos, strings e números de ponto flutuante
- Verificação de tipos em operações aritméticas e lógicas
- Tratamento de erros de tipo através de mônadas

2 Arquitetura do Compilador

O compilador foi estruturado em módulos bem definidos, seguindo a arquitetura clássica de compiladores:

2.1 Análise léxica

A análise léxica foi implementada utilizando Alex, gerador de analisadores léxicos para Haskell. O arquivo `SlLexer.x` define as regras lexicais da linguagem SL.

2.1.1 Tokens implementados

Os seguintes grupos de tokens foram definidos:

```haskell
data Lexeme
    = TLet | TAssign | TAttribution | TIdent { out :: String }
    | TLParen | TRParen | TSemi | TComma | TFunc   | TStruct
    | TIf | TElse | TWhile | TReturn | TLBrace | TRBrace
    | TLBracket | TRBracket  | TPrint | TRead | TDot
    -- type tokens
    | TInt | TFloat | TBool | TVoid | TString
    -- literal tokens
    | LitInt Int | LitFloat Double | TTrue | TFalse
    | LitString { out :: String }
    -- operators tokens
    | TPlus | TMinus | TTimes | TDiv | TMod
    -- relational operators tokens
    | TNot | TEq | TNeq | TLt | TGt | TLeq | TGeq | TAnd | TOr
     -- end of file token
    | TEOF
    deriving (Eq, Ord, Show)
```

Listing 1: Definição de tokens

2.1.2 Características do analisador léxico

- **Comentários**: Suporte a comentários de linha (`//`) e comentários de bloco (`/* */`)
- **Estados**: Implementação de estados para reconhecimento de declarações de variáveis e comentários
- **Literais**: Reconhecimento de literais inteiros, float, strings e booleanos
- **Operadores**: Conjunto completo de operadores aritméticos, lógicos e relacionais
- **Tratamento de erros**: Detecção de tokens inválidos e comentários não fechados

2.1.3 Estados do lexer

O analisador utiliza três estados principais:

- Estado inicial (0): reconhecimento de tokens gerais
- `state_string`: para reconheciomento de literais de string.
- `comment_state`: para comentários de bloco

2.1.4 Desafios

O reconhecimento de strings foi feito inicialmente por um padrão regex, mas ao realizar os testes este método se demonstrou bem limitado. Então busquei na internet alguma solução já implementada utilizando o wrapper monadUserState do Alex. Nessa busca encontrei o seguinte projeto, do qual utilizei a parte do código que faz reconhecimento das strings:

https://github.com/haskell/alex/blob/master/examples/tiger.x

2.1.5 Utilização de LLM

Para esta parte houve muito pouca utilização de Large Language Models, o uso ficou bastante restrito a correção de de código, a maior parte do desenvolvimento foi feito tomando como base código existentes e documentação das ferramentas.

2.2 Análise sintática

O parser foi implementado no arquivo SlParser.y usando o gerador Happy

2.2.1 Precedência e Associatividade

O parser define claramente a precedência dos operadores (do menor ao maior):

```
%right          '='             -- atribui  o (associativa    direita)
%left           '||'            -- ou l gico
%left           '&&'            -- e l gico
%nonassoc       '==' '!='       -- igualdade (n o associativo)
%nonassoc       '<' '>' '<=' '>='  -- relacionais
%left           '+' '-'         -- adi   o/subtra  o
%left           '*' '/' '%'     -- multiplica  o/divis o/m dulo
%right          '!'             -- nega  o
%left           '.' '[' ']' '(' ')'  -- acesso/chamada
```

Listing 2: Precedência e Associatividade

2.2.2 Árvore de sintaxe abstrata

A AST foi implementada no módulo `Sl.Frontend.Syntax.SlSyntax`

2.2.3 Ajuda de mecanismos de IA

Para começar o trabalho usei o código da Linguagem While disponível no material da disciplina, aqui houve muito auxílio de sugestão de código para adequar o código que já existia ao trabalho atual.

Até conseguir uma versão que funcionasse do Parser, não houve nenhum uso de LLM onde eu inseri um input pedindo por ajuda. Essa versão contava com 28 conflitos de shift/reduce e um conflito de reduce/reduce

Para resolver os conflitos utilizei recorri ao Claude Sonnet 4, os conflitos foram resolvidos atualizando as declarações de precedência, dividindo as expressões, que estavam todas definidas em Exp, em uma estrutura hierárquica

```
Exp : BinaryExp                     { $1 }

BinaryExp :
       UnaryExp                     { $1 }
     | BinaryExp '=' BinaryExp      { EAssign $1 $3 }
     | BinaryExp '==' BinaryExp     { $1 :=: $3 }
     | BinaryExp '!=' BinaryExp     { $1 :/=: $3 }
     | BinaryExp '<' BinaryExp      { $1 :<: $3 }
     | BinaryExp '>' BinaryExp      { $1 :>: $3 }
     | BinaryExp '<=' BinaryExp     { $1 :<=: $3 }
     | BinaryExp '>=' BinaryExp     { $1 :>=: $3 }
     | BinaryExp '&&' BinaryExp     { $1 :&: $3 }
     | BinaryExp '||' BinaryExp     { $1 :|: $3 }
```

```
      | BinaryExp '+' BinaryExp      { $1 :+: $3 }
      | BinaryExp '-' BinaryExp      { $1 :-: $3 }
      | BinaryExp '*' BinaryExp      { $1 :*: $3 }
      | BinaryExp '/' BinaryExp      { $1 :/: $3 }
      | BinaryExp '%' BinaryExp      { $1 :%: $3 }

UnaryExp :
        PrimaryExp                   { $1 }
      | '!' UnaryExp                 { ENot $2 }

-- Primary expressions and postfix operations
PrimaryExp :
        AtomExp                      { $1 }
      | PrimaryExp '[' Exp ']'       { EArrayAccess $1 $3 }      -- array
    access
      | PrimaryExp '.' var           { EFieldAccess $1 $3 }      -- field
    access

AtomExp :
        intLit                       { EValue (VInt $1) }
      | floatLit                     { EValue (VInt (round $1)) }  --
    temporary conversion
      | stringLit                    { EValue (VString $1) }
      | 'true'                       { EValue (VBool True) }
      | 'false'                      { EValue (VBool False) }
      | var                          { EVar $1 }
      | var '(' Args ')'             { ECall $1 $3 }
      | var '{' Args '}'             { EStructConstruct $1 $3 } -- struct
    construction
      | '(' Exp ')'                  { $2 }
```

Listing 3: Estrutura Hierárquica de Expressões

3 Resultados e Discussão

Por mais que o resultado entregado tenha sido longe do esperado, acredito que reflete mais a minha dificuldade de desenvolver no ambiente proposto do que a minha dedicação.

3.1 Instruções de Uso

Após acessar o container docker será necessário acessar o diretório do projeto sl, em seguida rodar o Análizador Léxico Alex, depois o gerador de parse happy e por fim compilar o projeto:

```
cd ./sl
alex ./src/Frontend/Lexer/SlLexer.x
happy ./src/Frontend/Parser/SlParser.y --ghc
cabal build
```

Para executar o compilador: A execução deve ser feita a partir do executável gerado, fornecendo uma opção de uso, (=–lexer= ou =–parser=)

```
./app/Main --[OPTION] ./examples/{1-6}.sl
```

3.2 Testes Realizados

Todos os testes fornecidos no enunciado foram feitos, mas não é esperado que o sexto teste funcione. Todos os outros testes devem funcionar, tanto para –lexer, quanto para –parser.

4 Conclusão

O projeto implementou uma base forte para um compilador para a linguagem SL usando Haskell com Alex/Happy. A arquitetura modular separa claramente lexer e parser, suportando funcionalidades essenciais como funções, estruturas, arrays e estruturas de controle. O sistema oferece visualização da AST, demonstrando uma aplicação prática bem-sucedida dos conceitos fundamentais de compiladores.

5 Referências

Referências

[1] The Alex and Happy team. *Alex 3.4.0.1 documentation*, 2024. Disponível em: `https://haskell-alex.readthedocs.io/en/latest/`. Acesso em: 20 de fevereiro de 2026.

[2] The Alex and Happy team. *Alex example — A lexer for the Tiger language*, 2023. Disponível em: `https://github.com/haskell/alex/blob/master/examples/tiger.x`. Acesso em: 20 de fevereiro de 2026.

[3] Hutton, G. (2016). *Programming in Haskell*. Cambridge University Press.

[4] Appel, A. W. (1998). *Modern Compiler Implementation in ML*. Cambridge University Press.

[5] WebAssembly Community Group. (2023). *WebAssembly Specification*.