Relatório de Projeto: Compilador SL

Hebert Luiz Madeira Pascoal
Matrícula: 23.1.4008

Victor Xavier Costa
Matrícula: 23.1.4003

BCC328 - Construção de Compiladores I - DECOM/UFOP

21 de fevereiro de 2026

Resumo

Neste trabalho prático nós construímos um compilador da linguagem SL para WebAssembly (WAT). Usamos o ambiente de desenvolvimento Haskell, com os pacotes Alex e Happy, para as análises léxicas e sintáticas, respectivamente. Neste documento nós documentamos as nossas escolhas e motivação por trás do resultado final.

Sumário

1 Introdução

A linguagem SL é uma linguagem fictícia[1] proposta no enunciado deste trabalho, que é inspirada em Rust[2]. Ela será uma linguagem de propósito geral compilada, imperativa, procedural, estático-tipada, de tipagem forte e com inferência de tipos de alto nível. Assim, em SL, pode-se definir funções de escopo global, e programas progressivamente mais complexos podem ser construídos a partir de composição de funções menores. A linguagem também permitirá tipos compostos (estruturas) e estrutura-de-dados indexadas (arrays).

2 Metodologia

Para o desenvolimento deste trabalho, utilizamos a linguagem Haskell com os frameworks Alex, para o desenvolvimento do analisador léxico, e Happy para a geração de um analisador sintático LALR(1) a partir da GLC definida. Nesta seção nós apresentaremos a linguagem SL, sua representação formal e definições estruturais em termos de implementação, utilizadas para estudarmos a sua análise.

2.1 Estrutura sintática de SL

A sintaxe de SL é bem parecida com a de Rust. Com o propósito de apresentarmos a sintaxe da linguagem, primeiramente olhemos para as palavras-chaves da linguagem (veja tab. 1).

Aqui estão alguns exemplos práticos da sintaxe da linguagem.

```
func factorial(n: int) : int {
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

func main(void) : int {
    let result : int = factorial(5);
    print(result); // deve imprimir 120
    return 0;
}
```

Listagem 1: Exemplo 1 SL. Fatorial de um inteiro, recursivo.

```
struct Person {
    name : string;
    age : int;
    height : float;
}

```

[1] Até onde se sabe; pesquisando, não achamos nada sobre ela.
[2] https://rust-lang.org/

Palavra-chave	Significado
func	Define uma função.
struct	Define uma estrutura.
let	Define uma variável.
return	Retorno de função.
if	Inicia um controle de fluxo.
elif	Alternativa de controle de fluxo.
else	Alternativa final de controle de fluxo.
for	Laço de repetição *for*.
while	Laço de repetição *while*.
forall	Declaração de tipos genéricos na definição de uma função.
new	Para a alocação de memória.
void	Tipo nulo, vazio.
bool	Tipo booleano.
int	Tipo de número inteiro.
float	Tipo de número flutuante.
string	Tipo de cadeia de caracteres.
true	Valor booleano verdadeiro.
false	Valor booleano falso.

Tabela 1: Palavras-chaves de SL.

```
7  func main() : void {
8      // arranjo
9      let people : Person[3];
10     people[0] = Person{ "Alice", 25, 1.65 };
11     people[1] = Person{ "Bob", 30, 1.80 };
12     people[2] = Person{ "Charlie", 35, 1.75 };
13
14     // it. sobre arranjo
15     let i : int = 0;
16     while (i < 3) {
17         print(people[i].name);
18         print(people[i].age);
19         print(people[i].height);
20
21         i = i + 1;
22     }
23 }
```

Listagem 2: Exemplo 2 SL. Estruturas e arranjos.

```
1  func reverse(arr : int[], size : int) : int [] {
2      let result : int[] = new int[size];
3
4      let i : int = 0;
5      while (i < size) {
6          result[i] = arr[size - i - 1];
7          i = i + 1;
```

```
    }

    return result;
}

func main() : void {
    let original : int[5] = [1, 2, 3, 4, 5];
    let reversed : int[] = reverse(original, 5);

    let j : int = 0;
    while (j < 5) {
        print(reversed[j]);
        j = j + 1;
    }
}
```

Listagem 3: Exemplo 3 SL. Programa para reverter arrays.

```
func calculateBMI(weight : float, height : float) : float {
    return weight / (height * height);
}

func isAdult(age : int) : bool {
    return age >= 18;
}

func main() : void {
    let bmi : float = calculateBMI(70.5, 1.75);
    let adult : bool = isAdult(20);

    print(bmi);
    print(adult);

    if (adult && bmi > 25.0) {
        print("adulto com sobrepeso");
    } else {
        print("condicao normal");
    }
}
```

Listagem 4: Exemplo 4 SL. Cálculo numérico em ponto flutuante.

```
func id(x) {
    return x;
}

forall a b . func map (f: (a) -> b, v: a[]) : b[] {
    let result = new b[v.size];

    for (i = 0; i < v.size; ++ i) {
        result[i] = f(v[i]);
    }

```

```
12      return result;
13  }
14
15  func a_real_nothing(x : int) : float {
16      if (x >= 5) {
17          return 1.0;
18      }
19
20      return 0.0;
21  }
22
23  func main(void) : void {
24
25      // identity~
26      print(id(5.0));
27      print(id(5));
28      print(id("ola"));
29
30      // map~
31      let asd : int[] = [ 3, 1, 4, 1, 5 ];
32      let new_asd : float[] = map(a_real_nothing, asd);
33
34      for (i : int = 0; i < new_asd.size; ++ i) {
35          print(new_asd[i]);
36      }
37  }
```

Listagem 5: Exemplo 5 SL. Inferência de tipo e Generics.

A partir disso, vejamos a gramática sintática geral da linguagem. Apresentaremos a gramática de forma top-down.

2.1.1 Gramática: Estrutura procedural geral

$$P \longrightarrow \lambda \mid D\ P \tag{1}$$
$$D \longrightarrow St \mid F \tag{2}$$
$$St \longrightarrow \textbf{struct}\ I\ \{\ \hat{V}\ \} \tag{3}$$
$$\hat{V} \longrightarrow \lambda \mid V\ \textbf{;}\ \hat{V} \tag{4}$$
$$F \longrightarrow \textbf{func}\ I(Pm)\ \ T_S^*\ \{\ \hat{C}\ \} \mid \textbf{forall}\ \hat{G}\ \textbf{.}\ \textbf{func}\ I(Pm)\ \ T_S^*\ \{\ \hat{C}\ \} \tag{5}$$
$$Pm \longrightarrow \lambda \mid \textbf{void} \mid Pm' \tag{6}$$
$$Pm' \longrightarrow V^* \mid V^*\ \textbf{,}\ Pm' \tag{7}$$
$$\hat{G} \longrightarrow I \mid I\ \hat{G} \tag{8}$$
$$\hat{C} \longrightarrow \lambda \mid C\ \hat{C} \tag{9}$$
$$C \longrightarrow Atr\ \textbf{;} \mid C_f \mid R \mid \textbf{return}\ E\ \textbf{;} \tag{10}$$
$$Atr \longrightarrow Atr_D \mid Atr_R \tag{11}$$
$$Atr_D \longrightarrow \textbf{let}\ V = E \mid \textbf{let}\ I = E \tag{12}$$
$$Atr_R \longrightarrow X_A = E \tag{13}$$
$$C_f \longrightarrow \textbf{if}\ (E)\ \{\ \hat{C}\ \}\ \overline{C_f} \tag{14}$$
$$\overline{C_f} \longrightarrow \lambda \mid \textbf{elif}\ (E)\ \{\ \hat{C}\ \}\ \overline{C_f} \mid \textbf{else}\ \{\ \hat{C}\ \} \tag{15}$$
$$R \longrightarrow \textbf{for}\ (E_R;\ E^*;\ E^*)\ \{\ \hat{C}\ \} \mid \textbf{while}\ (E)\ \{\ \hat{C}\ \} \tag{16}$$
$$E_R \longrightarrow Atr_R\ \mid\ V = E \mid E^* \tag{17}$$

2.1.2 Gramática: Expressões

Expressões representam um conjunto de operações sobre os objetos do programa que podem ser definidas inline. Estas operações vão desde de soma à aplicação de função. É virtuoso começarmos, então, listando as precedências de cada operador (veja tab. 2).

Operador	Precedência	Associatividade
$\|\|$	0	Direita
&&	1	Direita
== != < <= > >=	2	Não-associativo
+ −	3	Esquerda
* ÷ %	4	Esquerda
**	5	Direita
++ −−	6	Não-associativo

Tabela 2: Precedência e associatividade dos operadores. Quanto maior, maior a precedência.

Do que segue, a gramática para expressões fica[3]:

$$E \longrightarrow E_0 \tag{18}$$
$$E^* \longrightarrow \lambda \mid E \tag{19}$$
$$E_X \longrightarrow X \mid I\ (\ A\ ) \mid I\ \{\ A\ \} \mid \ [\ A\ ] \tag{20}$$
$$A \longrightarrow \lambda \mid A' \tag{21}$$
$$A' \longrightarrow E_X \mid E_X\ ,\ A' \tag{22}$$
$$E_0 \longrightarrow E_1 \mid E_0\ Op_0\ E_1 \tag{23}$$
$$E_1 \longrightarrow E_2 \mid E_1\ Op_1\ E_2 \tag{24}$$
$$E_2 \longrightarrow E_3 \mid E_2\ Op_2\ E_3 \tag{25}$$
$$E_3 \longrightarrow E_4 \mid E_3\ Op_3\ E_4 \tag{26}$$
$$E_4 \longrightarrow E_5 \mid E_4\ Op_4\ E_5 \mid UOp_3\ E_3 \tag{27}$$
$$E_5 \longrightarrow E_6 \mid E_5\ Op_5\ E_6 \tag{28}$$
$$E_6 \longrightarrow E_7 \mid E_7 \mid IncrOp\ E_6 \tag{29}$$
$$E_7 \longrightarrow E_X \mid \ (\ E_0\ ) \mid E_7\ IncrOp \tag{30}$$
$$Op_0 \longrightarrow \ \mid\mid \tag{31}$$
$$Op_1 \longrightarrow \&\& \tag{32}$$
$$Op_2 \longrightarrow == \mid !\,= \mid < \mid <= \mid > \mid >= \tag{33}$$
$$Op_3 \longrightarrow + \mid - \tag{34}$$
$$UOp_3 \longrightarrow + \mid - \tag{35}$$
$$Op_4 \longrightarrow * \mid \div \mid \% \tag{36}$$
$$Op_5 \longrightarrow ** \tag{37}$$
$$IncrOp \longrightarrow ++ \mid -- \tag{38}$$

Neste caso, resolveu-se a precedência expandindo-se as regras da expressão.

2.1.3 Gramática: Variáveis e Literais

$$V \longrightarrow I\ T_S \tag{39}$$
$$V^* \longrightarrow I\ T_S^* \tag{40}$$
$$T_S \longrightarrow :\ T \tag{41}$$
$$T_S^* \longrightarrow \lambda \mid T_S \tag{42}$$
$$X \longrightarrow X_A \mid L \tag{43}$$
$$X_A \longrightarrow I\ X_I' \tag{44}$$
$$X_A' \longrightarrow \lambda \mid I\ X_A' \mid \ .\ I\ X_A' \mid [\ E\ ]\ X_A' \tag{45}$$

Neste caso, X representa valor literal (L) ou referência X_A. Aqui, por referência entende-se "acesso à variável ou memória".

[3]Inspirada na tablea de precedência de C: Tabela de Precedência em C.

2.1.4 Gramática: Tipos

$$t \longrightarrow I \mid \textbf{int} \mid \textbf{float} \mid \textbf{string} \mid \textbf{bool} \tag{46}$$
$$T \longrightarrow t \mid t\ \hat{T}_I \mid (\ \hat{T}\ )\ \rightarrow\ T \tag{47}$$
$$\hat{T} \longrightarrow \lambda \mid \hat{T}' \tag{48}$$
$$\hat{T}' \longrightarrow T \mid \hat{T}'\ \textbf{,}\ T \tag{49}$$
$$\hat{T}_I \longrightarrow [E^*] \mid \hat{T}_I\ T_I \tag{50}$$
$$T_I \longrightarrow [\ E\ ] \tag{51}$$
$$\tag{52}$$

2.1.5 Gramática: Básico

$$I \longrightarrow \sigma I' \tag{53}$$
$$I' \longrightarrow \lambda \mid \sigma I' \mid \pi I' \tag{54}$$
$$\pi \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \tag{55}$$
$$\sigma \longrightarrow a - z \mid A - Z \mid \_ \tag{56}$$
$$L \longrightarrow b \mid i \mid f \mid s \tag{57}$$
$$b \longrightarrow \textbf{true} \mid \textbf{false} \tag{58}$$
$$i \longrightarrow \text{INTEGER} \tag{59}$$
$$f \longrightarrow \text{FLOAT} \tag{60}$$
$$s \longrightarrow \text{STRING} \tag{61}$$
$$\tag{62}$$

Aqui, as regras i, f e s expandem conforme suas expressões regulares.

Símbolo(s)	Significado	Regras
P	Programa. Variável de partida.	1
D	Definição (global).	2
F	(Def. de) Função.	5
Pm, Pm'	Parâmetros de função.	6, 7
$\hat{G}$	Sequência de variáveis de tipo: I^+.	8
$\hat{C}$	Sequência de comandos.	5, 9, 14, 15
C	Comando (instrução dentro do escopo de função.)	10, 9
Atr	Comando de atribuição.	10
Atr_D	Comando de atribuição por definição.	11
Atr_R	Comando de (re-)atribuição.	11
$\underline{C_f}$	Controle de fluxo. "Ifs".	10
$\overline{C_f}$	(Contra) Controle de fluxo. "Elses".	14, 14
R	Laços de repetição.	10
St	(Def. de) Estruturas	2
$\hat{V}$	Sequência de declaração de variáveis.	3, 4
E_R	Comando de início do laço **for**.	16, 18, 17
E	Expressão.	19, 16 17
E^*	Expressão opcionais (anulável).	16
A, A'	Lista de argumentos.	20 21, 22
E_X	Valor de expressão.	22
V	Declaração de variável.	4, 12, 17
V^*	Declaração de variável com especificação opcional de tipo.	7
T_S^*	Especificação de tipo (opcional).	5, 39
T_S	Especificação de tipo.	5, 39
X	Valor.	5, 39
X_A	Acesso de variável.	5, 39
L	Valor literal.	5, 39
I	Identificador, ou nome dos símbolos.	5, 8, 12
t	Tipo base.	46, 47
T	Tipo.	47, 49
$\hat{T}, \hat{T}'$	Lista de tipos.	47, 49, 49
$\hat{T}_I$	Lista de índices de tipos.	47, 50
T_I	Índice de tipo.	50

Tabela 3: Resumo da notação na gramática.

2.2 Sistema de tipos

Aqui discutiremos como os tipos foram organizados, em Haskell, para o desenvolvimento deste trabalho.

2.2.1 Tokens

Para começar, com respeito ao token, temos dois tipos; a saber:

```
data Token = Token {
    pos :: (Int, Int),
    lexeme :: Lexeme
} deriving (Eq, Ord, Show, Read)
```

```
data Lexeme =
    -- palavras chave.
    T_Func |
    ...

    -- tipos palavra-chave.
    T_TypeVoid |
    ...

    -- identificadores e literais.
    T_Identifier String |
    T_Integral Integer |
    ...

    -- fechamento.
    T_LParenthesis |
    ...

    -- operadores
    T_Plus
    ...

    -- outros
    T_Dot |
    T_Comma |
    ...

    T_EOF
```

Um token é uma composição de uma posição e um lexema; a posição é um tupla de dois inteiros indicando a linha e a coluna, respectivamente. O lexema representa uma estrutura léxica da linguagem.

2.2.2 Árvore Sintática

Para a árvore de sintaxe abstrata, define-se os seguintes tipos auto-explicativos:

```
data IR_Program = Program [IR_Statement]

data IR_Statement = FuncDef { ... } | StructDef { ... }
```

```
data IR_Command =
    VarDef IR_Var IR_Expression |
    Assignment IR_VarAccess IR_Expression |
    Return IR_Expression |
    If IR_Expression [IR_Command] [IR_Command] |
    While IR_Expression [IR_Command] |
    For IR_Command IR_Expression IR_Expression [IR_Command] |
    CmdExpression IR_Expression

data IR_Var = VarDecl Identifier IR_Type

data IR_VarAccess =
    VarAccess Identifier IR_VarAccess |
    VarAccessIndex IR_Expression IR_VarAccess |
    VarAccessNothing

data IR_Type = TypeVoid | TypeBool | TypeInt | TypeFloat |
   TypeString | TypeArray IR_Type [IR_Expression] | TypeFunction
   IR_Type IR_Type | TypeGeneric Identifier

data IR_Expression =
    ExpNothing |
    ExpVariable     IR_VarAccess |
    ExpLitInteger   Integer |
    ExpLitFloating  Double |
    ExpLitBoolean   Bool |
    ExpLitString    String |

    ExpSum          IR_Expression IR_Expression |
    ExpSub          IR_Expression IR_Expression |
    ExpMul          IR_Expression IR_Expression |
    ExpDiv          IR_Expression IR_Expression |
    ExpIntDiv       IR_Expression IR_Expression |
    ExpMod          IR_Expression IR_Expression |
    ExpPow          IR_Expression IR_Expression |
    ExpNegative     IR_Expression |
    ExpAnd          IR_Expression IR_Expression |
    ExpOr           IR_Expression IR_Expression |
    ExpEq           IR_Expression IR_Expression |
    ExpNeq          IR_Expression IR_Expression |
    ExpGt           IR_Expression IR_Expression |
    ExpGeq          IR_Expression IR_Expression |
    ExpLt           IR_Expression IR_Expression |
    ExpLeq          IR_Expression IR_Expression |
    ExpLIncr IR_Expression |
    ExpRIncr IR_Expression |
    ExpLDecr IR_Expression |
    ExpRDecr IR_Expression |

    ExpFCall Identifier [IR_Expression] |
```

```
    ExpStructInstance Identifier [IR_Expression] |
    ExpArrayInstancing [IR_Expression] |
    ExpNew IR_Type
```

A composição de tais tipos possibilita a construção de uma árvore de derivação completa durante a análise sintática, conforme apresentaremos adiante (3.3).

2.2.3 Lexer

A seguinte função para tokenização da string do programa é definida:

```
lexer :: String -> Either String [Token]
```

A primeira entrada do `Either` representa erro e, a segunda, a lista de tokens tokenizados. Por praticidade (principalmente para os testes), define-se também

```
lexer_plain :: String -> [Lexeme]
lexer_plain s = case lexer s of
    Left _          -> []
    Right tokens    -> map lexeme tokens
```

2.2.4 Parser

O parser é:

```
parse_sl_alex :: Alex IR_Program
parse_sl_exp_alex :: Alex IR_Expression

parse_sl :: String -> Either String IR_Program
parse_sl_exp :: String -> Either String IR_Expression
```

onde `Alex` é uma instancia de mônada definida pelo gerador de analisador léxico Alex. Nesta definição, `parse_sl` é a função que opera a análise sintática da lista de tokens retornadas pela chamada da análise léxica sobre a string de entrada. A primeira entrada do `Either` é representa o erro propagado como uma string, e a segunda, a árvore sintática construída.

Aqui, `parse_sl_exp` foi também definida para a simplificação dos testes direcionados especificamente à análise sintática de expressões.

2.2.5 Pretty

Para a representação legível da árvore sintática obtida após o parsing dos programas de entrada, define-se a operação pretty. O pretty é definido como uma classe de tipo que define a seguinte função:

```
class Pretty t where
    pretty :: t -> PrettyContext ()
```

onde `PrettyContext` é tipo que define o contexto de operações pretty, para o qual são definidas as suas instâncias de funtor, funtor aplicativo, e mônada. Essencialmente, `PrettyContext` define uma transição de estados que mantém a informação do nível de indentação corrente para a construção da string de saída. O valor propagado nas transições do pretty context é uma tupla que informa o estado atual, a string construída e o restante da entrada a ser processada.

```
newtype PrettyContext t = PC {
    pc_transition :: PrettyState -> (PrettyState, String, t)
}

data PrettyState = PrettyState {
    identation_level :: Int
}
```

2.2.6 Erros

A fim de garantir diagnósticos mais precisos e facilitar a depuração dos códigos SL compilados, definimos uma separação lógica dos possíveis erros gerados durante a compilação. Cada tipo de erro definido corresponde ao momento em que foi gerado segundo o pipeline do compilador, conforme pode-se observar:

```
newtype SrcPos = SrcPos (Int, Int)

data ErrorType =
    LexicalError |
    SyntaxError |
    SemanticalError


data Error = Error {
    error_type :: ErrorType,
    error_msg :: String,
    error_pos :: SrcPos
}
```

Como é de se esperar, foram também definidas as instâncias de `Show` para cada um deles, na intenção de garantir maior qualidade para as mensagens de erro devolvidas para o usuário.

2.3 Inferência de tipos

Pendente ...

3 Arquitetura do Compilador

O projeto do compilador construído foi denominado `HarmonicalVortex` e está, até então, estruturado da seguinte maneira:

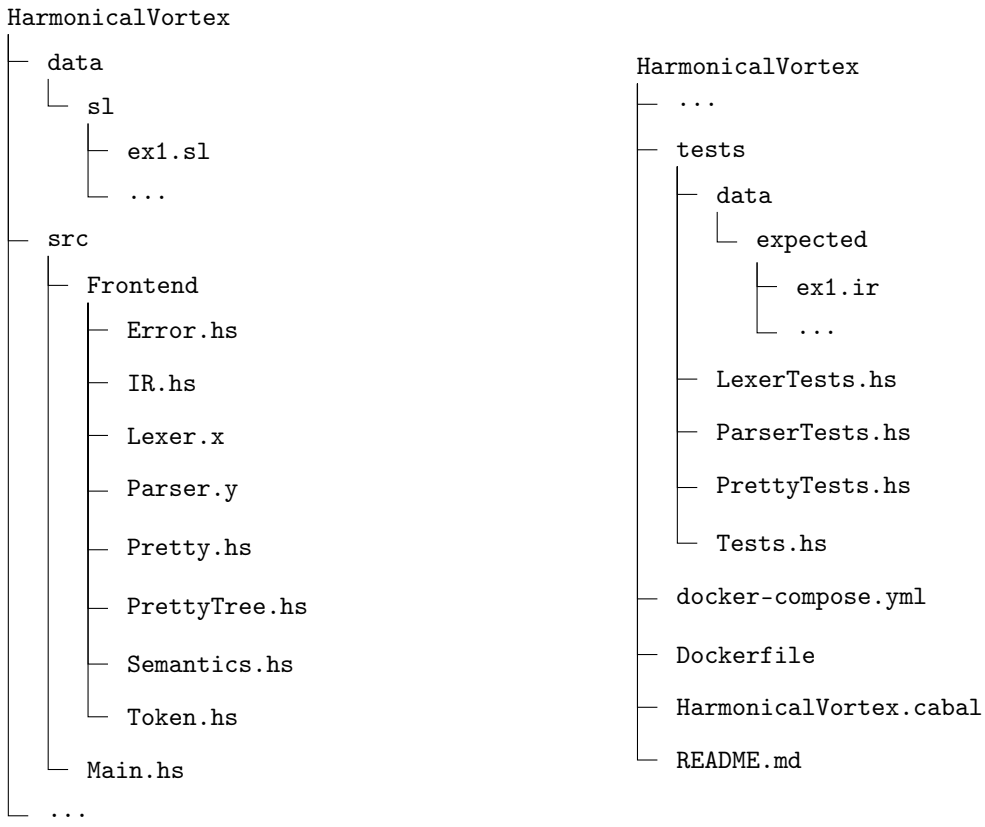

Figura 1: Estrutura de diretórios do projeto HarmonicalVortex

O projeto está organizado seguindo uma simples divisão lógica: dados de entrada; código-fonte do compilador; e um módulo de testes 4.5. No que tange à implementação do compilador, o projeto foi modularizado considerando a separação entre o frontend e backend do compilador. Inicialmente, trabalhamos apenas sobre o frontend, que constitui toda tarefa relativa à análise do código fonte da linguagem, sendo essencialmente divido em: Análise Léxica, Sintática e Semântica.

Dentro do frontend, em `Error.hs`, separamos logicamente os diferentes tipos de erro que o compilador pode emitir, na intenção de facilitar a depuração dos códigos dos usuários, conforme descrito em 2.2.6. `IR.hs` define os tipos que compõe a representação intermediária ou árvore de sintaxe do compilador (veja 2.2.2). `Token.hs`, por sua vez, armazena as definiçõs dos tipos Token e Lexeme 2.2.1. `Lexer.x` e `Parser.y` apresentam a implementação do analisador léxico pelo framework do Alex, e do analisador sintático definido pelo Happy, respectivamente. Esses arquivos são compilados para código Haskell pelos drivers destes frameworks produzindo, no mesmo diretório, os arquivos `Lexer.hs` e `Parser.hs`. Por fim, em `Pretty` e `PrettyTree` as definições descritas em 2.2.5 são aplicadas para a construção de uma string "embelezada" da árvore de sintaxe obtida na análise léxica.

Futuramente, em `Semantics.hs`, será implementado o analisador semântico de SL.

3.1 Análise léxica

Conforme mencionado, o analisador léxico foi construído segundo as especificações do Alex[4]. Especificamente, utilizamos o wrapper `"monadUserState"` que que combina execução monádica com a possibilidade de manter um estado definido pelo usuário durante a análise léxica (`AlexUserState`). Escolhemos este wrapper justamente para tirar proveito desta definição de estado, sendo que acabamos aproveitando esta habilidade apenas para que o analisador léxico possibilitasse o aninhamento de comentários, uma característica não exigida para a linguagem, mas considerada útil. No entanto, consideramos ainda que tal habilidade pode ser útil para outros fins, caso necessário em avaliações futuras.

No geral, a arquitetura do analisador léxico segue as especificações do framework do Alex. Primeiro, definimos as expressões regulares básicas para dígitos, números inteiros, números em ponto flutuante, caracteres, identificadores e literais string. Depois, definimos as regras de produção dos diferentes tokens baseado nas ER's definidas. Por fim, especificamos os procedimentos básicos de erro, manipulação do estado de usuário definido e a chamada principal do analisador léxico em si (veja 2.2.3).

3.2 Análise sintática

Para análise sintática utilizamos o Happy. Seu framework básico é um gerador de analisadores sintáticos do tipo LALR(1).

Na configuração do Happy, primeiro especificamos as definições básicas do analisador: o tipo mônada utilizado (quando deseja-se definir um parser monádico), a chamada do analisador léxico utilizado para a produção dos tokens (no caso, especificamos o tipo da mônada do Alex e o analisador léxico construído anteriormente). Em seguida, definimos os símbolos terminais da gramática e as regras para extração dos mesmos, o que corresponde a um simples casamento de padrão com os tipos de Token que os representa. Então, definimos a precedência e associatividade dos diferentes operadores de acordo com a sintaxe do Happy: os operadores devem ser especificados na ordem de menor para maior precedência com os rótulos de associatividade respectivos (`%left`, `%right`, ou `%nonassoc`). Seguindo, especificamos as regras de derivação que definem a gramática livre-de-contexto que produz a linguagem SL, conforme especificado em 2.1. Neste passo, tenta-se uma tradução fiel da gramática, contudo seja possível observar pequenas divergências em alguns pontos na implementação por questões práticas. Finalmente, especificamos a chamada principal para o parser.

3.3 Árvore de sintaxe abstrata

A seguir, apresenta-se o esquema que representa um exemplo prático da estrutura da árvore de sintaxe abstrata gerada a partir da função fatorial definida no exemplo 2, segundo as definições es Haskell apresentadas em 2.2.2.

[4]Documentação do Alex

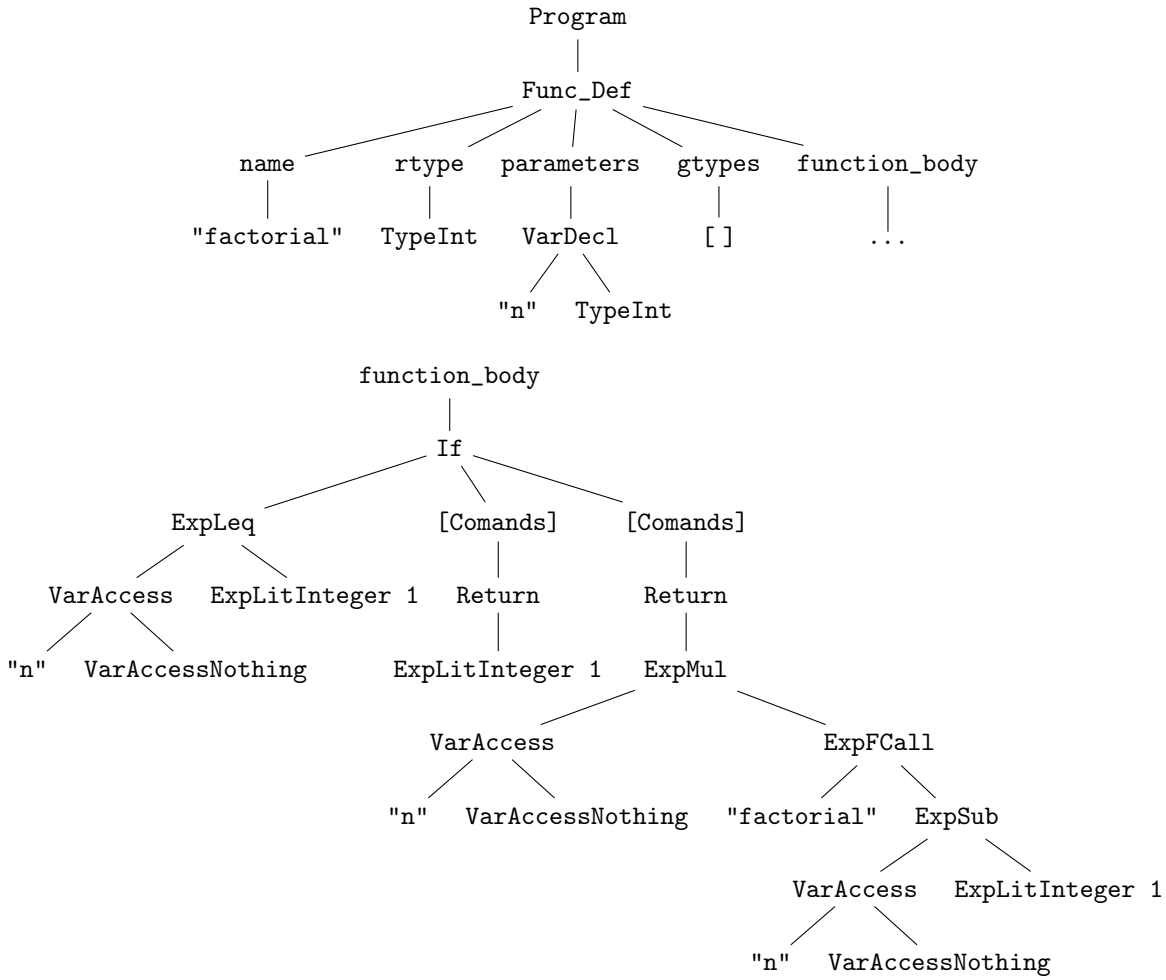

Figura 2: Exemplo de árvore de sintaxe obtida pelo parsing da função fatorial.

4 Resultados e Discussão

4.1 Backlog da Etapa 1

Para a primeira etapa do projeto, foi desenvolvido a gramática formal da linguagem SL, bem como os analisadores léxico e sintático. Utilizamos os frameworks Happy e Alex de Haskell.

Ao final desta etapa, obtivemos os seguintes recursos funcionando conforme especificado no enunciado:

- Analisador léxico completo e funcionando devidamente para todos os exemplos apresentados.
- Analisador sintático também completo, funcionando para todos os códigos de exemplo apresentados, de tal modo que a gramática definida não apresenta nenhum conflito do tipo *shift-reduce* ou *reduce-reduce* para o analisádor LALR(1) do Happy.
- Módulo de *pretty-print* tanto para a reconstrução do programa a partir da IR, quanto da estrutura de dados em si conforme definida em Haskell.
- Testes automatizados para os analisadores léxico e sintático, bem como para o módulo pretty.
- Sistemas de tipos de erro funcionando parcialmente.

4.2 Backlog da Etapa 2

Na segunda etapa do projeto, foram desenvolvidos tanto o analisador semântico quanto o interpretador para a linguagem. Estes módulos foram

Ao final dessa etapa, a análise semântica e o interpretador apresentavam os seguintes recursos funcionando:

- Reconhecimento de execução correta de todos os exemplos apresentados e para alguns exemplos adicionais. [5]
- Reconhecimento e interpretação de expressões lambdas a partir da técnica de *lambda-lifting* durante a análise semântica.
- Inferência de tipos para funções genéricas e/ou com tipos não anotados (por meio da monorfisação), sem conversão implícita para tipos mais gerais.
- Reconhecimento e interpretação da função de saída `print`.
- Recurso básico de otimização: Redução de expressões simples durante análise semântica. (Ex.: `a = 2 * 3 + 5` $\rightarrow$ `a = 11`).
- Recursos adicionais para a gramática adicionados conforme a necessidade observada durante o desenvolvimento da análise semântica (Como por exemplo suporte a expressões lambdas).

[5] No estado final apresentado no repositório, o exemplo 5 (`data/sl/ex5.sl`) e o exemplo 6 (adicional) passaram a falhar nos testes de interpretação por um problema de integração das funções lambda com o sistema de monomorfismo implementado. No entanto, em estados anteriores, quando não aplicávamos o monomorfismo conseguimos interpretá-las sem problemas

- Testes Automatizados para o analisador sintático e interpretador.

Observações:

- As funções de entrada e saída `print` e `scan` foram implementadas como símbolos especiais da linguagem que são artificialmente inseridos no tabela de símbolos para do programa analisado durante a análise semâtica. No entanto, apenas a função `print` é suportada na interpretação.

4.3 Backlog da Etapa 3

Para a terceira etapa, não foi possível desenvolvermos a geração de código WebAssembly de programas completos, mas apenas um *setup* com a definição da estrutura planejada para tradução da IR obtida da análise semântica a partir da qual construiríamos a representação textual em formato `.wat` a ser convertida no bytecode Wasm.

Desse modo, objetivamente, temos os seguintes recursos implementados:

- Estrutura de dados representacional de código WebAssembly.
- Módulo *pretty* para a construção de código em formato `.wat`.

Infelizmente, não foi possível construir a geração de código para um programa SL completo, porém o restante da implementação (conversão da IR na estrutura de código WASM) poderia ser concluída de modo caso houvesse um pouco mais de tempo para o desenvolvimento do trabalho.

4.4 Divisão de Tarefas

Durante o desenvolvimento do projeto, as principais da tarefas realizadas foram dividas da seguinte forma:

- **Implementação dos Analisadores Léxico e Sintático:** Desenvolvidos em *pair-programming.*
- **Implementação da Análise Semântica**: Hebert
- **Implementação do Interpretador**: Victor
- **Implementação do Setup para a Geração de Código**: Desenvolvidos em *pair-programming.*

Vale ressaltar, que apesar da divisão apresentada, todos os autores estiveram envolvidos durante de alguma forma durante as implementações de cada etapa do compilador.

4.5 Testes Automatizados

Os testes realizados foram estruturados utilizando o framework `Test.Hspec`, onde cada cenário de teste deve ser definido como uma função monádica do tipo `Spec` que posteriormente podem ser instanciadas na função `hspec`. Assim, definimos os seguintes módulos de teste: `LexerTests`, `ParserTests`, `PrettyTests`, onde são definidos os testes do analisador léxico, sintático e do modo de impressão "pretty", respectivamente.

```
main :: IO ()
main = hspec $ do
    LexerTests.tests
    ParserTests.tests
    PrettyTests.tests
```

Em cada função de teste, definimos um item de teste com a função `it`, onde especificamos a operação a ser executada e o resultado esperado, comparando-os com a função `shouldBe`, conforme o seguinte exemplo:

```
cflx_specs :: Spec
cflx_specs = describe "Parsing Control Flux" $ do
    it "only if" $ do
        let parsed = parse_sl "func main() : void { if (true) {} }"
        parsed `shouldBe` (Right $ Program [FuncDef {
            function_name       = "main",
            function_rtype      = TypeVoid,
            function_parameters = [],
            function_gtypes     = [],
            function_body       = [
                If (ExpLitBoolean True) [] []
            ]
        }])
```

os testes desenvolvidos incluem:

- Casos de erro (léxico, sintático e semântico)
- Representação de literais numéricos
- Expressões
- Definições de structs.
- Definições de funções.
- Controle de fluxo.
- Arranjos.
- Testes léxicos, sintáticos, semânticos e de interpretação sobre arquivos de entrada.

Instruções de Uso

Conforme as especificações deste trabalho, o projeto desenvolvido deve ter como plataforma-alvo o ambiente docker baseado em linux-ubutu:22.04 cujas dependências e configurações foram pré-estabelecidas de acordo com o arquivos `Dockerfile` e `docker-compose` fornecidos. Com base nessa premissa, apresentamos as seguintes instruções para a utilização do compilador desenvolvido:

Execução do ambiente docker compose

```
    make run-docker
```

Ou, alternativamente:

```
    docker-compose up -d
    docker-compose exec sl bash
```

Construção do projeto

```
cabal build
```

Execução do compilador

```
cabal run HarmonicalVortex -- <arquivo sl> [opcoes]
```

com as seguintes opções disponíveis:

- `-l, --lexer`: Executa somente a análise láxica e exibe os tokens.
- `-p, --parser`: Executa a análise léxica e sintática, exibindo a árvore de sintaxe obtida.
- `-pt, --pretty`: Executa a análise léxica e sintática, exibindo uma versão textual do programa reconstruído através da árvore de sintaxe obtida.
- `-s, --semantics`: Executa a análise semântica sobre o resultado da análise semântica.
- `-i, --interpreter`: Executa o interpretador sobre o resultado da análise semântica.

Observações: A opção `-c, --compile`: foi polanejada para executar integralmente o compilador gerando código webassembly, porém no estado atual apenas executa um *pretty* sobre uma estrutura de código construída manulamente para teste.

Execução dos testes automatizados

```
cabal test
```

5 Conclusão

Neste trabalho desenvolvemos, inicialmente, um analisador léxico e um analisador sintático para a linguagem SL. Utilizamos os frameworks Alex e Happy para a geração destes analisadores, para os quais foi necessário definir os lexemas e tokens pertencentes à linguagem, bem como a gramática livre de contexto relativa à mesma. Durante a construção dos analisadores foram consideradas questões como os requisitos explícitos da linguagem; características não especificadas, mas desejadas; e o escopo do trabalho. Através dos testes definidos, conseguimos avaliar que os analisadores construídos se mostraram efetivos na tarefa de tokenizar e fazer o parsing de diversos programas exemplo da linguagem SL.

Referências

[1] Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.

[2] Marlow, S., Gill, A., et al. *Happy: The Parser Generator for Haskell — User Guide*. Disponível em: `https://www.haskell.org/happy/`.

[3] Marlow, S., Gill, A., et al. *Alex: The Lexical Analyser Generator for Haskell — User Guide*. Disponível em: `https://www.haskell.org/alex/`.

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