RELATÓRIO DO TRABALHO 02
ANÁLISE SINTÁTICA LALR E BASEADA EM PEG E INTERPRETAÇÃO

BCC328 - Construção de compiladores I

Autores:
20.2.4182 - Leandro Marcos Mendes Zanetti
20.1.4016 - Pedro Parentoni de Almeida
Semestre 24.2

1 Introdução

Este relatório aborda detalhadamente o projeto em Haskell que se propõe a implementar um compilador para a linguagem chamada "lang". O projeto tem quatro componentes principais: A análise Léxica, a representação recursiva da árvore de Sintaxe e as análises sintáticas de LALR e PEG.

2 Contexto Geral do Projeto

O projeto visa simular o funcionamento de um compilador, abordando etapas fundamentais do processo de compilação. Como por exemplo:

1. Analisar a entrada fornecida por arquivos de código-fonte.
2. Reconhecer tokens utilizando um analisador léxico (lexer).
3. Construir uma árvore sintática que represente a estrutura hierárquica do código com diferentes análises sintáticas.
4. Exibir o resultado de forma legível no terminal.

Essas funcionalidades cobrem aspectos fundamentais da compilação, desde a identificação de palavras-chave e símbolos até a construção de uma representação estruturada que pode ser utilizada em fases posteriores, como a geração de código intermediário ou final.
O projeto contou com a utilização de algumas ferramentas, como o 'Alex' e o 'Happy':

- Alex: Gera automaticamente um analisador léxico a partir de regras definidas em um arquivo .x.
- Happy: Gera automaticamente um analisador sintático LALR a partir de regras gramaticais definidas em um arquivo .y.

3 Estrutura do projeto

O projeto está organizado em três principais diretórios: app, src e test.

1. Pasta app: Contém o arquivo principal Main.hs, responsável por iniciar o programa e integrar os módulos.
2. Pasta src: Contém os módulos principais do sistema:
    - Lexer.x: Arquivo com as definições e regras de todos os tokens da linguagem.
    - Lexer.hs: Implementação do analisador léxico que converte o código de entrada em uma lista de tokens. Ele identifica palavras-chave, identificadores, operadores e símbolos de pontuação da linguagem Lang. Esse arquivo foi gerado utilizando a ferramenta 'Alex' a partir do conteúdo de Lexer.x.
    - Parser.y: Arquivo com a definição e regras da gramática de Lang, além da especificação das regras de formação da Árvore de Sintaxe Abstrata (AST). Esse arquivo foi grado utilizando utilizando a ferramenta 'Happy'.

- Parser.hs: Responsável pela análise sintática LALR e geração da árvore sintática.
- RecursiveParser.hs: Implementa um analisador sintático descendente recursivo para a linguagem Lang. Alternativa ao LALR, feito manualmente, sem ferramentas automáticas.
- Interpreter.hs: Executa a AST gerada pelo parser, interpretando o código da linguagem Lang.
- Syntax.hs: Define algumas regras da gramática, como as expressões e a estrutura de um programa em Lang.
- PEG.hs: Responsável pela análise sintática, tem como objetivo executar o código representado pela árvore de sintaxe produzida pelo parser usando o interpretador.
- Simplecombinators.hs: Biblioteca de combinadores disponibilizada pelo professor.

3. Pasta test: Inclui arquivos de entrada como exemplo.lang, que servem como base para testar as funcionalidades do compilador.

4 Como Compilar e Executar

4.1 Compilar

```
stack build
```

4.2 Executar

```
stack exec -- lang-compiler-exe --<opção> test/<arquivo.lang>
```

4.3 Exemplo de Execução

```
stack exec -- lang-compiler-exe --lexer test/exemplo.lang
```

4.4 Dentro do Programa

É possível escolher diversos arquivos de teste, todos se encontram dentro da pasta test. As opções disponíveis:

- `exemplo.lang`
- `small.lang`
- `arvore.lang`
- `fatorial.lang`
- `fibonacci.lang`
- `ordenar.lang`
- `pilha.lang`

5 Arquivo Principal

O módulo `Main.hs` é responsável por gerenciar as funcionalidades do compilador. Ao ser executado, o compilador lê os comandos que foram passados como parâmetro para sua execução e seleciona as chamadas de funções respectivas desse comando. A principal função do módulo é servir de controlador entre a entrada do usuário, via prompt de comando, e a execução das funcionalidades do compilador, como a análise léxica e a análise sintática.
A implementação atual é funcional e oferece uma boa base para futuras extensões do compilador. Por exemplo, seria possível adicionar mais modos de operação, como um modo de geração de código intermediário ou final. Funcionalidades essas, que, serão colocadas em prática na próxima etapa do trabalho.

5.1 Fluxo de Execução

O fluxo de execução do módulo `Main.hs` é baseado em chamadas de funções de acordo com o que for escrito no prompt de comando pelo usuário. Com base no que é passado, o programa tem um comportamento distinto:

- No modo `lexer`, o programa lê o arquivo de entrada e verifica seu conteúdo, após isso, passa para a função responsável pela análise léxica, que irá gerar e exibir os tokens encontrados, ou seja, irá gerar uma lista de tokens. O lexer divide o código-fonte em unidades significativas, como palavras-chave, operadores e identificadores.
- No modo `recursive-tree`, o programa lê o arquivo de entrada e verifica seu conteúdo, após isso, passa esse conteúdo para o analisador sintático descendente recursivo, que processa os tokens e gera a árvore sintática correspondente. A árvore gerada representa a estrutura hierárquica do programa e é fundamental para as fases posteriores do compilador, como a geração de código intermediário.
- No modo `lalr`, o programa lê o arquivo de entrada e verifica seu contéudo, após isso, utiliza o parser gerado pela ferramenta 'Happy' e raliaza a análise sintática e gera a árvore correspondente.
- No modo `peg`, o programa lê o arquivo de entrada e verifica seu contéudo, após isso, executa o código representado pela árvore de sintaxe produzida pelo parser e usando o interpretador, lê o conteúdo da árvore.
- No modo `help`, o programa exibe informações explicativas sobre o funcionamento do compilador e as opções do menu.

Caso o usuário escolha uma opção inválida, o programa solicita mostra quais são as opções válidas disponíveis.

6 Lexer

O módulo Lexer.hs é gerado pela ferramenta 'Alex' a partir do conteúdo de `Lexer.x`, esse conteúdo conta com regras e definições dos tokens, como pro exemplo:

- Identifica palavras-chave como print, if, else, return.

- Reconhece símbolos de operação (+, -, *, /, etc.) e pontuação (;, , , etc.).
- Gera uma lista de tokens que servem como entrada para a próxima etapa.

A análise léxica é a primeira etapa no processo de compilação, onde o código-fonte é transformado em uma sequência de tokens. A função principal do módulo, lexer, recebe os tokens lidos pela ferramenta 'Alex' para serem apresentados, separadamente, ao susuário.

6.1 Definição do Tipo de Token

O arquivo `Lexer.x` conta com a definição do tipo de token, que representa as diferentes categorias de tokens reconhecidas pelo lexer. Cada token pode ser um identificador, como por exemplo: (`ID String`), um número inteiro (`INT Int`), ou uma palavra-chave, como `PRINT`, `IF`, `ELSE`, `THEN`, `RETURN`. Além disso, o lexer também reconhece operadores (`OP String`) e pontuação (`PUNC String`), como parênteses e operadores aritméticos.

```
data Token
  = ID String
  | TYPEID String
  | INT Int
  | FLOAT Float
  | CHAR Char
  | BOOL Bool
  | NULL
  | PRINT
  | IF
  | ELSE
  | THEN
  | RETURN
  | FUN
  | DATA
  | MAIN
  | ITERATE
  | READ
  | NEW
  | OP String
  | PUNC String
  | ASSIGN
  | WHILE
  | BREAK
  deriving (Show, Eq)
```

Cada construtor de `Token` representa uma categoria específica, e a função `Show` é derivada para permitir a exibição dos tokens no terminal durante a execução do programa.

6.2 Reconhecimento dos Tokens

Além de definir os Tokens, existe também as expressões regulares associadas a tipos de tokens da linguagem Lang. para reconhecimento desses tokens, isso significa, ler o arquivo

de entrada e saber definir quais são os tokens e atribui-los aos tipos previamente definidos, ou seja, o analisador léxico percorre caractere por caractere e, com base nas características de cada caractere, decide qual token gerar. Alguns exemplos:

```
tokens :-
[ \t\n\r]+               ;  -- Ignorar espaços e quebras de linha
$alpha($alnum)* { \s -> mkIdent s }  -- Identificador ou palavra-chave
$digit+        { \s -> INT (read s) }  -- Número inteiro
$digit+"."$digit+ { \s -> FLOAT (read s) }  -- Número float
"'[^\']'"      { \s -> case s of
                            ('\'':c:'\'':[]) -> CHAR c
                            _ -> error ("Invalid char literal: " ++ s) }
"&&"           { \_ -> OP "&&" }
"=="           { \_ -> OP "==" }
"!="           { \_ -> OP "!=" }
"<="           { \_ -> OP "<=" }
">="           { \_ -> OP ">=" }
"<"            { \_ -> OP "<" }
">"            { \_ -> OP ">" }
.
.
.
```

O analisador léxico começa verificando se o caractere atual é um espaço, um identificador, um número ou um símbolo de pontuação/operador. Se o caractere for um espaço, ele é ignorado e o analisador continua com os caracteres restantes. Se for um caractere alfabético `alpha` (Representa letras e underscore (a-zA-Z)), o analisador identifica palavras-chave ou identificadores como ID. Se o caractere for numérico `digit` (Representa dígitos numéricos (0-9)), o analisador processa o número e executa a conversão necessária, sendo inteiro ou ponto flutuante. Caso o caractere seja um dos símbolos de pontuação ou operadores definidos, o token correspondente é gerado. Se o caractere não se encaixar em nenhuma dessas categorias, um erro é gerado indicando um caractere inesperado.

7 ParserRecursive

O módulo `ParserRecursive.hs` é responsável pela analise sintática descendente recursiva do compilador "lang". Ele recebe uma sequência de tokens, gerada pelo analisador léxico (lexer), e constrói a Árvore de Sintaxe Abstrata (AST) que representa a estrutura hierárquica, a qual facilita a compreensão e o processamento do código-fonte. É importante ressaltar que o grupo teve muita dificuldade nesta etapa de parsing, além das dificuldades encontradas na implementação, houve problemas com a identação no momento da impressão da árvore no terminal. A seguir, será apresentada uma descrição detalhada sobre as principais funções presentes neste módulo. Ele utiliza técnicas de correspondência de padrões e funções auxiliares para processar estruturas como funções, blocos e expressões. A `AST` gerada é formatada e impressa na tela.
Definição do tipo de Árvore de Sintaxe Abstrata:

```
data AST
    = Node String [AST] -- Um nó com um rótulo e subárvores
    | Leaf String       -- Uma folha com um valor
    deriving (Show)
```

7.1 Definição da Árvore de Sintaxe (AST)

A estrutura principal do módulo é a Árvore de Sintaxe (AST), que é definida como um tipo algébrico com duas variantes:

- **Node**: Representa um nó da árvore que possui um rótulo (uma string) e uma lista de subárvores (ou filhos).
- **Leaf**: Representa uma folha da árvore que contém um valor simples (também uma string).

7.2 Função Principal: `parseAndPrintAST`

```
parseAndPrintAST :: String -> IO ()
parseAndPrintAST content = do
  let ast = parseProgram content
  putStrLn "\nÁrvore de Sintaxe Gerada:"
  putStrLn (formatAST ast)
```

Essa função é responsável por:

1. Tokenizar o conteúdo da entrada usando `lexer`.
2. Construir a `AST` utilizando `buildAST`.
3. Exibir a árvore formatada com `formatAST`.

7.3 Função `formatAST`

Essa função exibe a `AST` em formato estilizado como uma árvore hierárquica.

```
formatAST :: AST -> String
formatAST ast = unlines (formatHelper ast "")
```

Descrição do Auxiliar Que Existe Dentro da Função:

- `formatHelper`: Processa recursivamente os nós da `AST`, adicionando prefixos que representam a hierarquia.

7.4 Funções de Parsing

As funções a seguir processam `Token`s e constroem a `AST`.

7.4.1 Função `parseBlock`

```
parseBlock :: [Token] -> ([AST], [Token])
parseBlock (PUNC "{" : rest) =
  let (blockTokens, remaining) = span (/= PUNC "}") rest
  in (buildAST blockTokens, drop 1 remaining)
parseBlock tokens = ([], tokens)
```

`parseBlock` processa blocos delimitados por chaves ({ }), retornando a `AST` do bloco de código com seus respectivos tokens, além dos `Token`s restantes.

7.4.2 Função `parseExpression`

```
parseExpression :: [Token] -> ([AST], [Token])
parseExpression (ID name : rest) = ([Leaf ("ID: " ++ name)], rest)
parseExpression (INT val : rest) = ([Leaf ("INT: " ++ show val)], rest)
parseExpression (OP op : rest) = ([Leaf ("OP: " ++ op)], rest)
parseExpression (PUNC p : rest) = ([Leaf ("PUNC: " ++ p)], rest)
parseExpression tokens = ([], tokens)
```

Essa função identifica diferentes tipos de expressões (`ID`, `INT`, `OP`, etc.) e constrói folhas da `AST` para cada uma delas.

7.4.3 Função `buildAST`

```
buildAST :: [Token] -> [AST]
buildAST [] = []
buildAST (ID name : PUNC "(" : rest) =
  let (params, remaining1) = span (/= PUNC ")") rest
      (body, remaining2) = parseBlock (drop 1 remaining1)
  in Node ("Function: " ++ name) (map (Leaf . show) params ++ body) :
  buildAST remaining2
  ...
```

Essa função é maior dentro do arquivo "ParserRecursive.hs", para esse relatório foi selecionado apenas uma parte dela para explicação, por isso o uso das reticências, essa função é responsável por construir a `AST` completa com base em uma sequência de `Token`s. Os principais casos incluem:

- `Funções`: Identificadas por um `ID` seguido de parênteses, com parâmetros e corpo processados como nós e subárvores.
- `Print`: Cria um nó "Print" com subárvores representando as expressões.
- `If`: Constrói uma árvore com nós "If", "Then" e "Else", processando os respectivos blocos.
- `Return`: Constrói um nó "Return" com a expressão associada.
- `Outros casos`: Blocos são representados por nós "Block".

8 Parser LALR

O arquivo `Parser.y` define o **analisador sintático LALR** da linguagem lang usando a ferramenta 'Happy'. Ele recebe os tokens do analisador léxico (`Lexer.x`) e constrói a **Árvore de Sintaxe Abstrata (AST)** do programa.

8.1 Estrutura do Arquivo

8.1.1 Definição de Tokens

Os tokens são definidos com **nomes simbólicos** e associados aos valores do `Lexer.x`.
Exemplos de Tokens:

- **Palavras-chave**: `PRINT`, `IF`, `ELSE`, `RETURN`, `FUN`, `WHILE`.
- **Tipos de dados**: `INT`, `FLOAT`, `CHAR`, `BOOL`, `NULL`.
- **Operadores**: `OP`, `ASSIGN`, `OP_GE (>=)`, `OP_LE (<=)`.
- **Símbolos de pontuação**: `P_SEMICOLON (;)`, `P_COMMA (,)`, `P_LPAREN (()`, `P_RBRACE (})`.

8.1.2 Definição da Gramática (Regras Sintáticas)

A gramática é composta por **regras que estruturam o código** da linguagem lang.
Principais Regras:

- **Program** → Define a estrutura do programa (`main` e funções).
- **Declaration** → Declaração de funções (`fun nome(params) {}`).
- **Expression** → Expressões matemáticas e booleanas.
- **Statement** → Estruturas de controle (`if-else`, `while`, `return`, `print`).
- **Block** → Blocos { } que agrupam comandos.

- **Assignment** → Atribuição de valores (`x = 10;`).
- **ParamListWithTypes** → Lista de parâmetros de funções com tipos.

8.1.3 Construção da AST (Árvore de Sintaxe Abstrata)

A AST é representada por um tipo Haskell, onde cada **nó** representa uma **estrutura do programa**.
Exemplos de nós na AST:

- `FunStmt` → Representa uma **declaração de função**.
- `IfStmt` → Representa um **comando if-else**.
- `WhileStmt` → Representa um **loop while**.
- `OpExpr` → Representa uma **operação matemática** (`a + b`).

8.1.4 Função de Impressão da AST

- A função `prettyPrintAST` gera uma **representação hierárquica** da AST.
- Usa **indentação** para facilitar a leitura.

Exemplo de Impressão da AST:

```
L MainProgram
    L FunStmt: fat
    |   L Params: [ID: num :: Int]
    |   L ReturnTypes: [Int]
    |   L Body:
    |       L IfStmt
    |       |   L Condition: (num < 1)
    |       |   L Then: return 1
    |       |   L Else: return num * fat(num-1)
```

9 Interpreter

O arquivo `Interpreter.hs` implementa um **interpretador** para a linguagem lang, processando a **AST** gerada pelo parser e executando os comandos. O interpretador processa expressões, variáveis e fluxos de controle dentro de um ambiente de execução estruturado, avalia todo o conteúdo de entrada e verifica se conseguiu percorrer por toda a gramática sem encotrar erros, exibindo uma mensagem de sucesso ao fim da verificação.

9.1 Principais Componentes

- **Env** – Representa o ambiente de execução (variáveis e valores).
- **interpOp** - Abstrai operações matemáticas para inteiros e floats.Aplica a operação correta com base nos tipos dos valores fornecidos. Se os operandos não forem compatíveis, gera um erro de tipo.

- **interpExp** – Avalia expressões, incluindo operações matemáticas.
- **interpStmt** – Processa comandos como atribuições, `print`, `if-else` e `return`.
- **interpBlock** – Executa blocos de código sequencialmente.
- **interpProgram** – Inicia a execução do programa.

9.2 Funcionamento

- Avalia expressões e operadores matemáticos.
- Gerencia variáveis e blocos de código.
- Executa comandos de entrada/saída e controle de fluxo.

10 Syntax

O arquivo `Syntax.hs` define a **estrutura de dados** usada na representação da linguagem lang, convertendo a **AST** gerada pelo parser em um formato estruturado, permitindo a conversão da AST em uma representação estruturada que pode ser processada pelo interpretador.

10.1 Principais Componentes

- **Value** – Representa valores primitivos (`Int`, `Float`, `Bool`, `Char`).
- **Exp** – Estrutura de expressões matemáticas e lógicas.
- **Ty** – Define os tipos suportados pela linguagem.
- **Program** – Representa um programa contendo declarações.
- **Decl** – Define funções com nome, parâmetros, tipo de retorno e corpo.
- **Block** – Conjunto de comandos de um escopo.
- **Stmt** – Estruturas de controle como `if`, `while`, atribuições e chamadas de função.

10.2 Conversão da AST

- **astToProgram** – Converte a AST para um `Program`.
- **astToDecl** – Processa declarações de funções.
- **astToBlock** – Converte blocos de código.
- **astToStmt** – Transforma comandos em estrutura interpretável.
- **astToExp** – Converte expressões da AST para sua forma manipulável.
- **strToTy** – Mapeia strings para tipos definidos na linguagem.

11 PEG

Nesta etapa o grupo encontrou muitas dificuldades. O arquivo `PEG.hs` tenta implementar a análise sintática e a interpretação de código da linguagem lang utilizando **Parsing Expression Grammar (PEG)**. Ele tenta realizar a análise sintática da entrada fornecida, gera a árvore de sintaxe abstrata (AST) e executa o código interpretado.

11.1 Módulos Importados

Para realizar suas funções, `PEG.hs` importa os seguintes módulos:

- **Lexer** – Utiliza `alexScanTokens` para converter a entrada em tokens lexicais.
- **Parser** – Emprega `parser` para gerar a AST a partir dos tokens.
- **Syntax** – Contém `astToProgram`, que converte a AST na estrutura do programa.
- **Interpreter** – Utiliza `interpProgram` para interpretar e executar o programa representado pela AST.

11.2 Função Principal: `parsePEGAndInterpret`

A função principal, `parsePEGAndInterpret`, executa o pipeline de análise e execução do código Lang. Seus passos são:

1. **Tokenização**: Converte a entrada bruta em tokens usando `alexScanTokens`.
2. **Análise Sintática**: Aplica o `parser` para construir a AST.
3. **Conversão**: Usa `astToProgram` para transformar a AST na estrutura de programa.
4. **Execução**: Interpreta e executa o programa via `interpProgram`.

O grupo teve bastante dificuldade nesta etapa do PEG, não foi possível realizar a conversão do código-fonte para sua forma estruturada, imporssibilitando sua execução por meio do interpretador, o que o grupo conseguiu fazer foi ler o arquivo de entrada e rodar o interpretador, passando por todo seu conteúdo e encerrando sem mensagens de erros, a impressão da árvore foi feita utilizando a função presente no parser do LALR.

12 Teste de Entrada e Saída

12.1 Entrada

O programa abaixo é um exemplo de uso do compilador, ele está alocado dentro da pasta "test" e serve para testar todo o projeto:

```
main() {
  print fat(10)[0];
}

fat(num :: Int) : Int {
```

```
  if (num < 1)
    return 1;
  else
    return num * fat(num-1)[0];
}

divmod(num :: Int, div :: Int) : Int, Int {
  q = num / div;
  r = num % div;
  return q, r;
}
```

12.2 Saída

12.2.1 Saída do Lexer

Abaixo tem-se um exemplo de saída que o compilador fornece, quando se utiliza a opção "Lexer":

```
Tokens encontrados:
ID "main"
PUNC "("
PUNC ")"
PUNC "{"
PRINT
ID "fat"
PUNC "("
INT 10
PUNC ")"
PUNC "["
INT 0
PUNC "]"
PUNC ";"
PUNC "}"
ID "fat"
PUNC "("
ID "num"
PUNC ":"
PUNC ":"
ID "Int"
PUNC ")"
PUNC ":"
ID "Int"
PUNC "{"
IF
PUNC "("
ID "num"
```

```
OP "<"
INT 1
PUNC ")"
RETURN
INT 1
PUNC ";"
ELSE
RETURN
ID "num"
OP "*"
ID "fat"
PUNC "("
ID "num"
OP "-"
INT 1
PUNC ")"
PUNC "["
INT 0
PUNC "]"
PUNC ";"
PUNC "}"
ID "divmod"
PUNC "("
ID "num"
PUNC ":"
PUNC ":"
ID "Int"
PUNC ","
ID "div"
PUNC ":"
PUNC ":"
ID "Int"
PUNC ")"
PUNC ":"
ID "Int"
PUNC ","
ID "Int"
PUNC "{"
ID "q"
OP "="
ID "num"
OP "/"
ID "div"
PUNC ";"
ID "r"
```

```
OP "="
ID "num"
OP "%"
ID "div"
PUNC ";"
RETURN
ID "q"
PUNC ","
ID "r"
PUNC ";"
PUNC "}"
```

12.2.2 Saída do ParserRescursive

Devido aos problemas e dificuldades com a implementação e depois com a identação da árvore, o compilador "entrega" como resultado uma árvore com os tokens do programa lido em sequência, buscando obedecer a hierarquia entre eles.
A árvore para o programa de entrada `exemplo.lang` foi construída e teve como saída:

```
|- Program
|  |- Function: main
|  |  |- Print
|  |  |  L- ID: fat
|  |  L- PUNC "("
|  |  L- INT 10
|  |  L- PUNC ")"
|  |  L- PUNC "["
|  |  L- INT 0
|  |  L- PUNC "]"
|  |  L- PUNC ";"
|  |- Function: fat
|  |  L- ID "num"
|  |  L- PUNC ":"
|  |  L- PUNC ":"
|  |  L- ID "Int"
|  L- PUNC ":"
|  L- ID "Int"
|  |- Block
|  |- If
|  |  L- PUNC: (
|  |  |- Then
|  |  |- Else
|  L- ID "num"
|  L- OP "<"
|  L- INT 1
|  L- PUNC ")"
```

```
|  |- Return
|  |  L- INT: 1
|  L- PUNC ";"
|  L- ELSE
|  |- Return
|  |  L- ID: num
|  L- OP "*"
|  |- Function: fat
|  |  L- ID "num"
|  |  L- OP "-"
|  |  L- INT 1
|  L- PUNC "["
|  L- INT 0
|  L- PUNC "]"
|  L- PUNC ";"
|  L- PUNC "}"
|  |- Function: divmod
|  |  L- ID "num"
|  |  L- PUNC ":"
|  |  L- PUNC ":"
|  |  L- ID "Int"
|  |  L- PUNC ","
|  |  L- ID "div"
|  |  L- PUNC ":"
|  |  L- PUNC ":"
|  |  L- ID "Int"
|  L- PUNC ":"
|  L- ID "Int"
|  L- PUNC ","
|  L- ID "Int"
|  |- Block
|  L- ID "q"
|  L- OP "="
|  L- ID "num"
|  L- OP "/"
|  L- ID "div"
|  L- PUNC ";"
|  L- ID "r"
|  L- OP "="
|  L- ID "num"
|  L- OP "%"
|  L- ID "div"
|  L- PUNC ";"
|  |- Return
|  |  L- ID: q
```

```
|  L- PUNC ","
|  L- ID "r"
|  L- PUNC ";"
|  L- PUNC "}"
```

12.2.3 Saída do Parser LALR

Ainda utilizando o arquivo de exemplo `exemplo.lang`:

```
    L- ProgramStmtList
    L- FunStmt
        L- Name: [ID: main]
        L- Params: []
        L- ReturnTypes: []
        L- BlockStmt
            L- PrintStmt
                L- IndexExpr
                    L- Array:
                        L- CallExpr
                            L- Name: [ID: fat]
                            L- Args:
                                L- IntExpr [INT: 10]
                    L- Index:
                        L- IntExpr [INT: 0]
    L- FunStmt
        L- Name: [ID: fat]
        L- Params: [ID: num :: Int]
        L- ReturnTypes: ["Int"]
        L- BlockStmt
            L- IfStmt
                L- Condition:
                    L- OpExpr [OP: <]
                        L- Left:
                            L- VarExpr [ID: num]
                        L- Right:
                            L- IntExpr [INT: 1]
                L- Then:
                    L- ReturnStmt
                        L- IntExpr [INT: 1]
                L- Else:
                    L- ReturnStmt
                        L- OpExpr [OP: *]
                            L- Left:
                                L- VarExpr [ID: num]
                            L- Right:
                                L- IndexExpr
```

```
                                L- Array:
                                    L- CallExpr
                                        L- Name: [ID: fat]
                                        L- Args:
                                            L- OpExpr [OP: -]
                                                L- Left:
                                                    L- VarExpr [ID: num]
                                                L- Right:
                                                    L- IntExpr [INT: 1]
                                L- Index:
                                    L- IntExpr [INT: 0]
L- FunStmt
    L- Name: [ID: divmod]
    L- Params: [ID: num :: Int] [ID: div :: Int]
    L- ReturnTypes: ["Int","Int"]
    L- BlockStmt
        L- AssignmentStmt
            L- LValueAssignment
                L- LValue:
                    L- VarExpr [ID: q]
                L- Expr:
                    L- OpExpr [OP: /]
                        L- Left:
                            L- VarExpr [ID: num]
                        L- Right:
                            L- VarExpr [ID: div]
        L- AssignmentStmt
            L- LValueAssignment
                L- LValue:
                    L- VarExpr [ID: r]
                L- Expr:
                    L- OpExpr [OP: %]
                        L- Left:
                            L- VarExpr [ID: num]
                        L- Right:
                            L- VarExpr [ID: div]
        L- ReturnStmt
            L- VarExpr [ID: q]
            L- VarExpr [ID: r]
```

13 Desafios Encontrados

1. O grupo continuou com dificuldades na impressão da árvore do analisador sintático descendente recursivo.

2. Densolvimento do PEG: O grupo não conseguiu executar o PEG da forma que se esperava, teve que adaptar com funções do LALR, o grupo tentou utilizar as bibliotecas do PEG disponibilizadas pelo professor no repositório da disciplina mas não obteve sucesso na adaptação para lang. A saída produzida pelo PEG é a mesma árvore de sintaxe produzida pelo LALR, por esse motivo não foi inserido o resultado da saída gerada pelo PEG, pois são iguais. Após a criação da árvore pelo PEG, o interpretador recebe a árvore como entrada e realiza o processo de interpretação. A grande dificuldade nesta etapa foi de fato desenvolver o PEG, o grupo não conseguiu entregar da maneira correta.

3. Geração e Visualização da árvore: Sobre o analisador sintático descendente recursivo, o grupo não conseguiu fazer a árvore ficar corretamente identada e bem apresentável na tela, houveram diversas tentativas sem sucesso, porém, para o LALR foi possível construir a aŕvore sem problemas de identação, pois com o uso da ferramenta 'Happy' a geração do `Parser.hs` facilitou a construção da árvore.

14 Recomendações de Melhoria

1. Tratamento de Erros: Adicionar mensagens de erro mais descritivas no parser para auxiliar na depuração.

2. Desenvolver o PEG: Entender melhor como se dá o desenvolvimento do PEG a partir das bilbliotecas já prontas e entregá-lo com suas funcionalidades esperadas na próxima etapa do trabalho, já que nessa versão, o PEG não ficou completo e funcional da forma correta.

15 Conclusão

O projeto do compilador "lang" foi desenvolvido com o objetivo de implementar as fases iniciais de um compilador simples, consistindo principalmente da análise léxica e da análise sintática. O módulo de análise léxica (`Lexer.hs`), responsável por identificar e classificar os tokens de um programa, juntamente com o módulo de análise sintática (`Parser.hs`), que constrói a Árvore de Sintaxe Abstracta (AST) a partir dos tokens gerados, formam a base para a transformação do código-fonte em uma representação estruturada que pode ser utilizada em etapas posteriores, como a geração de código intermediário ou otimizações.

15.1 Análise Léxica

A função de análise léxica (`lexer`) tem um papel crucial em reconhecer a linguagem do código-fonte. Ela é responsável por processar a entrada caractere por caractere, agrupando-os em tokens, que são as unidades de significado para o compilador. O módulo trata de diferentes tipos de tokens, como identificadores, números, operadores e pontuações, assegurando que o código seja segmentado de forma adequada para a análise sintática subsequente. Com a utilização da ferramenta 'Alex' a análise léxica fica mais direta do que fazer "a mão" e possibilita um melhor entendimento de como é gerado o arquivo .hs a partir do arquivo .x onde está situado todas as regras e definições.
A principal vantagem dessa abordagem é a clareza e modularidade do código, o que facilita a manutenção e a expansão. A implementação do lexer permite uma rápida identificação de erros de sintaxe simples, como caracteres inesperados. Além disso, a inclusão de palavras-chave como `if`, `else`, `then` e `print` no lexer permite que o compilador compreenda e trate essas construções de forma apropriada.

15.2 Análise Sintática

Apesar dessa etapa ter proporcionado diversas dificuldades e impossibilidades, é importante entender seu funcionamento teórico: O código desenvolvido do analisador sintático descendente recursivo implementa um parser simples e funcional para a construção de uma Árvore de Sintaxe Abstrata (AST), a partir de uma sequência de tokens gerada pelo analisador léxico. Esse parser foi projetado para identificar estruturas sintáticas específicas, como blocos de código delimitados por chaves, expressões matemáticas e estruturas condicionais, representando-as de forma hierárquica por meio de nós e folhas na AST.
Além disso, para o analisador sintático LALR, o grupo conseguiu ver melhor como se dá a construção do arquivo de parser gerado pela ferramenta 'Happy', que também facilita a construção do projeto ao invés de digitar todo o arquivo manualmente. Para o PEG, onde o grupo teve mais dificuldades, ficou mais difícil de visualizar a forma correta desse analisador, espera-se conseguir uma evolução para a próxima entrega do trabalho ter uma versão do PEG que funcione como o esperado, diferentemente da versão do trabalho atual.
Os analisadores sintáticos que o grupo tentou desenvolver, um analisador descendente recursivo, LALR e PEG, oferecem vantagens como a simplicidade de implementação, maior clareza no código e facilidade de depuração. Eles são especialmente úteis para linguagens com gramáticas simples, e como dito anteriormente, podem servir como base para futuras extensões, como análise semântica ou geração de código.