Documentação da Análise Sintática do Trabalho de Compiladores

Alunos: Ana Paula Carneiro Athayde, Lucas Thomas

Data: 25 de Outubro de 2023

Instituição: Universidade do Estado de Santa Catarina

Curso: Ciência da Compuação

Introdução

Este documento descreve a implementação da análise sintática para um compilador desenvolvido em Haskell, nele descrevemos a estrutura e o funcionamento do compilador, destacando os principais componentes e decisões de design. O trabalho foi realizado como parte da disciplina de Compiladores do Departamento de Ciência da Computação da UDESC, Joinville, na primeira fase do projeto. Na segunda fase será feita a análise semântica (verificação de tipos) e na terceira a geração de código a partir da representação intermediária.

O que é um Compilador?

Um compilador é um software que funciona como um tradutor. Imagine que você deseja escrever um programa em uma linguagem que você entende, como o inglês, para um computador que só entende uma linguagem muito básica e específica. O compilador é como um intérprete que traduz seu programa da sua linguagem para a linguagem do computador, tornando-o compreensível para a máquina. Ou seja, de maneira mais formal um compilador é um software que converte código-fonte escrito em uma linguagem de alto nível em código de máquina ou linguagem de baixo nível. Ele realiza essa tarefa em várias etapas sequenciais:

1. **Análise Léxica**: O compilador analisa o programa, identificando palavras-chave, nomes de variáveis, números e símbolos especiais. Ele os transforma em

"peças" chamadas tokens.

2. **Análise Sintática**: Depois de dividir o programa em tokens, o compilador verifica como essas partes se encaixam para formar uma estrutura coerente. É como verificar se todas as palavras em uma frase estão na ordem correta.
3. **Análise Semântica**: Aqui, o compilador verifica se o programa faz sentido. Ele verifica se você está usando as palavras e números corretos e se está fazendo operações compatíveis. Por exemplo, não faria sentido somar uma palavra a um número.
4. **Geração de Código Intermediário**: O compilador cria uma versão intermediária do programa, mais fácil de trabalhar. É como se você traduzisse uma história em um esboço simples antes de escrever um livro.
5. **Otimização (Opcional)**: Em algumas situações, o compilador faz melhorias para tornar o programa mais rápido ou usar menos memória.
6. **Geração de Código de Máquina**: Aqui, o compilador traduz o programa intermediário em uma linguagem que o computador realmente pode entender e executar. É como traduzir o esboço do livro para o idioma da impressora.
7. **Linkagem (Opcional)**: Se você tiver várias partes do programa, o compilador as une em um único programa.
8. **Código Executável**: Agora, você tem um programa que pode ser executado no computador para realizar tarefas específicas, como jogar um jogo, navegar na internet ou fazer cálculos.

O que é Haskell?

Haskell é uma linguagem de programação funcional pura que se destaca por sua elegância, expressividade e forte ênfase na imutabilidade e tipagem estática. Ela é conhecida por sua concisão e clareza na escrita de código.

- **Funcional**: Em Haskell, a programação é baseada em funções, e as funções são cidadãs de primeira classe. Isso significa que você pode passar funções como argumentos, retornar funções como resultados e criar funções de ordem superior.
- **Pura**: Haskell promove a imutabilidade e evita efeitos colaterais. Isso torna o código mais previsível e seguro.
- **Tipagem Estática**: Haskell é fortemente tipado, o que significa que você deve especificar os tipos das variáveis e o compilador verifica esses tipos em tempo

de compilação, evitando erros comuns.

- **Avaliação Preguiçosa**: Haskell utiliza uma estratégia de avaliação preguiçosa, o que significa que as expressões não são avaliadas até que seu resultado seja realmente necessário.

Aqui está um exemplo simples em Haskell e como executá-lo no terminal:

Exemplo Simples:

```
-- Definindo uma função para calcular o quadrado de um número
quadrado :: Int -> Int
quadrado x = x * x

-- Função principal que chama a função quadrado
main :: IO ()
main = do
    let numero = 5
    let resultado = quadrado numero
    putStrLn ("O quadrado de " ++ show numero ++ " é " ++ show resultado)
```

Como Rodar no Terminal:

1. Certifique-se de ter o compilador GHC (Glasgow Haskell Compiler) instalado no seu sistema. Você pode baixá-lo em https://www.haskell.org/ghc/.
2. Salve o código acima em um arquivo com extensão ".hs", por exemplo, "quadrado.hs".
3. Abra o terminal e navegue até o diretório onde você salvou o arquivo "quadrado.hs".
4. Compile o programa com o seguinte comando:

   ```
   ghc quadrado.hs
   ```

5. Isso criará um arquivo executável chamado "quadrado" no mesmo diretório.
6. Execute o programa com:

   ```
   ./quadrado
   ```

Você verá a saída no terminal, que neste caso será "O quadrado de 5 é 25". Esse é um exemplo simples de como escrever e executar um programa Haskell.

Estrutura do Código

A implementação da análise sintática é feita com o uso da biblioteca Parsec. A linguagem de origem possui suporte para três tipos de dados: `int`, `double` e `string`.

O Parsec é uma biblioteca em Haskell que permite criar analisadores de gramáticas de forma modular e expressiva, facilitando a definição de regras gramaticais de uma linguagem e construção de analisadores precisos. Ou seja, combinadores monádicos ajudam a lidar com análise de texto de maneira organizada e eficiente, sendo especialmente úteis na construção de compiladores e analisadores.

Produções da Gramática

O compilador segue as seguintes produções da gramática:

- `<Programa>` → `<ListaFuncoes>` `<BlocoPrincipal>`
- `<ListaFuncoes>` → `<Função>` `<ListaFuncoes>` | ε
- `<Funcao>` → `<TipoRetorno> id (<DeclParametros>) <BlocoPrincipal>`
- `<TipoRetorno>` → `<Tipo>` | void
- `<DeclParametros>` → `<Tipo>` id `<Parametros>` | ε
- `<Parametros>` → , `<DeclParametros>` | ε
- `<BlocoPrincipal>` → {<BlocoPrincipal'>}
- `<BlocoPrincipal'>` → `<Declaracoes>` `<ListaCmd>`
- `<Declaracoes>` → `<Tipo>` `<ListaId>` ; `<Declaracoes>` | ε
- `<Tipo>` → int | string | double
- `<ListaId>` → id `<ListaId'>`
- `<ListaId'>` → , `<ListaId>` | ε
- `<Bloco>` → { `<ListaCmd>` }
- `<ListaCmd>` → `<Comando>` `<ListaCmd>` | ε
- `<ChamadaFuncao>` → id (<ListaParametros>)
- `<ListaParametros>` → `<ListaParametros'>` | ε

- `<ListaParametros’>` → `<Expressao>` `<ListaParametros’’>`
- `<ListaParametros’’>` → , `<ListaParametros’>` | ε
- `<Comando>` → return `<TvzExpressao>`; | if (<ExpressaoLogica>) `<Bloco>` `<Senao>` | while (<ExpressaoLogica>) `<Bloco>` | id = `<Expressao>`; | print (<Expressao>); | read (id); | `<ChamadaFunção>`;
- `<TvzExpressao>` → `<Expressao>` | ε
- `<Senao>` → else `<Bloco>` | ε

Expressões e Operadores

- As expressões relacionais envolvem os operadores: <, >, <=, >=, ==, /=.
- As expressões lógicas envolvem os operadores: && (conjunção), || (disjunção) e ! (negação).
- A precedência dos operadores lógicos é definida como ! (maior), && (médio), || (menor).
- Os operadores aritméticos (+, -, *, /) seguem a precedência usual.
- As expressões podem conter identificadores de variáveis, constantes inteiras, constantes com ponto flutuante e chamadas de funções.
- Parênteses podem ser usados para alterar a ordem de avaliação.

Tokens

Os tokens identificador (id), constante inteira, constante com ponto flutuante e constante de cadeia de caracteres (literal) devem ser definidos conforme o padrão em linguagens de programação.

Representação Intermediária

A representação intermediária do compilador é definida pelos seguintes tipos algébricos de dados:

- `Id`: Representa um identificador.
- `Tipo`: Representa os tipos de dados (TDouble, TInt, TString, TVoid).
- `TCons`: Representa constantes (CDouble, CInt).
- `Expr`: Representa expressões aritméticas.
- `ExprR`: Representa expressões relacionais.

- `ExprL`: Representa expressões lógicas.
- `Var`: Representa variáveis.
- `Funcao`: Representa funções.
- `Programa`: Representa o programa completo.

Estrutura de Arquivos

O projeto do compilador está organizado em vários arquivos, cada um com uma função específica:

1. **Comandos.hs**: Contém definições e funções relacionadas à análise de comandos na linguagem.
2. **DataTypes.hs**: Define os tipos de dados utilizados no compilador, como `Tipo`, `TCons`, `Expr`, `ExprR`, `ExprL`, `Var`, `Funcao`, `Programa`, `Bloco` e `Comando`.
3. **Lex.hs**: Lida com a análise léxica e define os tokens da linguagem, como operadores e palavras-chave.
4. **Logico.hs**: Contém funções relacionadas à análise de expressões lógicas e operadores lógicos.
5. **Main.hs**: O arquivo principal que chama o analisador do compilador para processar o código-fonte.
6. **Makefile**: Um arquivo de script para compilar e executar o código.
7. **Parametro.hs**: Lida com a análise de parâmetros e declarações na linguagem.
8. **Programa.hs**: Contém definições e funções relacionadas à análise de funções e programas na linguagem.
9. **Relacional.hs**: Define operadores relacionais e funções relacionadas à análise desses operadores.
10. **Tabelas.hs**: Define tabelas de precedência para operadores aritméticos e lógicos, além de funções relacionadas à análise de tipos de dados.
11. **teste1.j--**: Um exemplo de código-fonte na linguagem a ser compilada.

Exemplo de Código

Aqui está um exemplo de código-fonte na linguagem que nosso compilador suporta:

```
double maior (double a, double b)
{
  int m;

  if (a > b) {m = a;}
  else {m = b;}
  return b;
}

int fat (int n)
{
  int f;

  f = 0;
  while (n > 0)
  {
    f = f * n;
    n = n - 1;
  }
  return f;
}


void imprimir(string s, double r)
{
  int s;

  print (s);
  print (r);
  return 0;
}

{
  int x, num;
  double a;
  int x;

  print("Numero:");
  read (num);
  x = fat (4);
  a = maior (2.5, 10);
  imprimir("teste:", 2);
  return 0;
}
```

Explicação dos Arquivo `.hs` :

Comandos.hs:

O arquivo `Comandos.hs` contém definições de comandos da linguagem, juntamente com funções para analisar e manipular esses comandos.

1. **Imports e Módulo**: O arquivo começa importando módulos necessários, como `Text.Parsec` para análise sintática, módulos relacionados à linguagem, como `DataTypes`, `Lex`, `Relacional`, `Tabelas`, `Parametro` e `Logico`, além de outros módulos que podem ser usados para análise sintática.
2. **Definição de Blocos**:
   - `blocoPrincipal`: Esta função analisa o bloco principal do programa, que é delimitado por chaves `{}`. Ela chama `bloco'` e concatena as declarações comandos.
   - `bloco'`: Essa função analisa blocos que podem conter declarações seguidas de comandos. Ela retorna uma tupla com as declarações e comandos.
3. **Comandos**:
   - `comandoif`: Analisa a estrutura condicional `if`. Pode incluir uma cláusula `else`. Retorna um valor do tipo `If`.
   - `comandowhile`: Analisa o comando `while`. Retorna um valor do tipo `While`.
   - `atribuicao`: Analisa atribuições, onde uma variável recebe um valor. Retorna um valor do tipo `Atrib`.
   - `comandoread`: Analisa o comando de leitura `read`, que lê um valor para uma variável. Retorna um valor do tipo `Leitura`.
   - `comandoprint`: Analisa o comando de impressão `print`. Retorna um valor do tipo `Print`.
   - `comandoreturn`: Analisa o comando `return`, que pode incluir ou não uma expressão de retorno. Retorna um valor do tipo `Ret`.
   - `comandocall`: Analisa chamadas de função. Pode ser uma atribuição ou apenas uma chamada de função. Retorna um valor do tipo `Proc`.
4. **`comando'`**: Esta função define a análise geral de comandos. Ela tenta analisar vários tipos de comandos, como condicionais, loops, atribuições, leitura, impressão, retorno e chamadas de função. O operador `<?>` é usado para fornecer uma mensagem de erro personalizada se a análise falhar.

Este arquivo é essencial para a análise sintática do compilador, uma vez que define como os comandos da linguagem são reconhecidos e estruturados.

DataTypes.hs:

O arquivo `DataTypes.hs` contém definições de tipos de dados e estruturas de dados que são usados em todo o projeto de compiladores.

1. **Definições de Tipos**:
   - `type Id = String`: Isso define um tipo de alias (sinônimo) chamado `Id`, que é uma representação de identificadores como strings.
   - `data Tipo = TDouble | TInt | TString | TVoid deriving Show`: Define um tipo de dados chamado `Tipo`, que representa os tipos de dados na linguagem. Os possíveis valores são `TDouble`, `TInt`, `TString` e `TVoid`.
   - `data TCons = CDouble Double | CInt Integer deriving Show`: Define um tipo de dados chamado `TCons`, que representa constantes na linguagem. Pode ser uma constante de ponto flutuante (`CDouble`) ou uma constante inteira (`CInt`).
2. **Expressões**:
   - `data Expr`: Define um tipo de dados chamado `Expr`, que representa expressões na linguagem. As expressões podem ser operações aritméticas (`:+:`, `:-:`, `:*:`, `:/:`), negações (`Neg`), constantes (`Const`), identificadores (`IdVar`), chamadas de função (`Chamada`) e literais (`Lit`).
3. **Expressões Relacionais e Lógicas**:
   - `data ExprR`: Define um tipo de dados chamado `ExprR`, que representa expressões relacionais na linguagem. Inclui operadores relacionais como igualdade (`:==:`), diferença (`:/=:`), menor que (`:<:`), maior que (`:>:`), menor ou igual a (`:<=:`) e maior ou igual a (`:>=:`).
   - `data ExprL`: Define um tipo de dados chamado `ExprL`, que representa expressões lógicas na linguagem. Pode ser uma conjunção (`:&:`), disjunção (`:|:`), negação (`Not`) ou uma expressão relacional (`Rel`).
4. **Variáveis e Funções**:
   - `data Var`: Define um tipo de dados chamado `Var`, que representa variáveis e seus tipos. Isso é usado para associar um identificador (`Id`) a um tipo (`Tipo`).
   - `data Funcao`: Define um tipo de dados chamado `Funcao`, que representa funções na linguagem. Ele inclui o nome da função (`Id`), a lista de parâmetros (`[Var]`) e o tipo de retorno (`Tipo`).
5. **Programa**:

- `data Programa`: Define um tipo de dados chamado `Programa`, que representa o programa completo. Ele inclui uma lista de funções (`[Funcao]`), uma lista de declarações de variáveis globais (`[Var]`) e o bloco principal do programa (`Bloco`).

6. **Bloco e Comandos**:
   - `type Bloco = [Comando]`: Define um tipo de alias chamado `Bloco`, que é uma lista de comandos.
   - `data Comando`: Define um tipo de dados chamado `Comando`, que representa comandos na linguagem. Isso inclui estruturas condicionais (`If`), loops (`While`), atribuições (`Atrib`), leituras (`Leitura`), impressões (`Imp`), retornos (`Ret`), chamadas de função (`Proc`), impressões (`Print`) e ações de leitura (`ReadAction`).

Essas definições de tipos de dados são fundamentais para representar a estrutura e semântica da da linguagem no compilador. Elas são usadas em toda a análise sintática, semântica e geração de código do projeto.

Lex.hs:

O arquivo `Lex.hs` contém definições relacionadas à análise léxica da linguagem que o compilador suporta. Ele define o léxico da linguagem, incluindo palavras-chave, operadores e símbolos.

1. **Importações e Módulo**:
   - O módulo `Lex` importa módulos necessários da biblioteca Parsec para a análise léxica.
2. **Definição do Léxico**:
   - `definicao`: Define configurações para o analisador léxico, incluindo definições de comentários, palavras reservadas e operadores. Isso é usado para configurar o analisador léxico da biblioteca Parsec.
   - `lexico`: Utiliza as configurações definidas em `definicao` para criar um analisador léxico com funções para analisar números, símbolos, parênteses, operadores, literais, identificadores e outros elementos da linguagem.
   - `numero`, `simbolo`, `parenteses`, `operator`, `literal`, `identifier`, `comando`, `reserved`, `braces`, `pontovirgula`: Estas funções são criadas com base no analisador léxico configurado com `lexico` e são usadas para analisar

elementos específicos, como números, símbolos, operadores, literais, identificadores, etc.

3. **Definição de Operadores**:
   - `prefix name fun`: Define um operador prefixo com um nome específico e a função correspondente para lidar com ele na análise sintática. Isso é usado para operadores como `!`.
   - `binario name fun`: Define um operador binário com um nome específico e a função correspondente para lidar com ele na análise sintática. Isso é usado para operadores binários, como `+`, `-`, `*`, `/`, `==`, `/=`, `<`, `<=`, `>`, `>=`, `&&`, `||`, `=`, entre outros.

Essas definições de léxico são essenciais para a análise léxica do compilador. Elas especificam como os tokens são reconhecidos e tratados, e são usadas em conjunto com as regras de análise sintática para compreender a estrutura da linguagem.

Logico.hs:

O arquivo `Logico.hs` contêm definições e funções relacionadas à análise de expressões lógicas na linguagem que o compilador suporta. Aqui estão as principais partes do código:

1. **Importações e Módulo**:
   - O módulo `Logico` importa módulos necessários da biblioteca Parsec para análise sintática.
   - Também importa módulos relacionados à linguagem, como `DataTypes`, `Lex`, `Relacional` e `Tabelas`.
2. **Expressões Lógicas**:
   - `exprL`: Esta função define a análise de expressões lógicas. Ela tenta analisar expressões lógicas entre parênteses (`parenteses logic`) ou expressões relacionais (`Rel`).
   - `logic`: Utiliza a função `buildExpressionParser` para construir um analisador de expressões lógicas com base em uma tabela de operadores definida em `tabelaL`. Essa função é usada para analisar expressões lógicas na linguagem.
3. **Expressões Gerais**:

- `expr`: Define a análise de expressões gerais, que podem ser usadas dentro de expressões lógicas. Ela utiliza a função `buildExpressionParser` para construir um analisador de expressões com base em uma tabela de operadores definida em `tabela`.

4. **Expressões Relacionais**:
   - `exprR`: Esta função analisa expressões relacionais, que envolvem operadores relacionais. Ela combina uma expressão (`expr`), um operador relacional (`opr`), e outra expressão (`expr`) para criar uma expressão relacional completa.
5. **Fatores**:
   - `fator`: Define a análise de fatores em expressões. Os fatores podem ser expressões entre parênteses, constantes, literais, chamadas de função ou identificadores.
   - `constant`: Analisa constantes, que podem ser números inteiros ou de ponto flutuante. Ela lida com números e decide se eles são inteiros ou de ponto flutuante.
   - `Literal` e `IdVar`: Analisam literais e identificadores, respetivamente.
6. **Constantes**:
   - `constant`: Define a análise de constantes. Ela analisa números, e dependendo do tipo (inteiro ou ponto flutuante), cria uma expressão constante correspondente.

Essas definições são parte da análise sintática da linguagem, particularmente da análise de expressões lógicas e relacionais. Elas são essenciais para entender a semântica e estrutura da linguagem.

Main.hs:

O arquivo `Main.hs` contém o código principal do compilador.

1. **Importações**:
   - O arquivo importa módulos necessários da biblioteca Parsec para análise sintática.
   - Também importa outros módulos relacionados ao projeto, como `DataTypes`, `Comandos` e `Programa`. Isso sugere que o código principal dependerá de definições e funções presentes nesses módulos.

2. **Função `parseProgram`**:
    - A função `parseProgram` é definida para analisar a entrada de código-fonte da linguagem. Ela recebe uma string como entrada e retorna um valor do tipo `Either ParseError Programa`. Isso significa que, se a análise for bem-sucedida, será retornado um `Right Programa`, representando a árvore sintática do programa. Caso contrário, um `Left ParseError` será retornado, indicando um erro na análise.
3. **Função Principal (`main`)**:
    - A função `main` é a função principal do programa. Ela faz o seguinte:
        - Lê o conteúdo de um arquivo chamado "teste1.j--", contendo o código-fonte da linguagem que será analisada.
        - Chama a função `parseProgram` para analisar o código lido.
        - Verifica se a análise foi bem-sucedida ou se ocorreu um erro.
        - Se ocorreu um erro, imprime o erro na saída padrão.
        - Se a análise foi bem-sucedida, imprime a representação da árvore sintática do programa.

No geral, o arquivo `Main.hs` define o ponto de entrada do compilador, que lê um arquivo de código-fonte, realiza a análise sintática e imprime a árvore sintática resultante ou mensagens de erro. Este é um componente essencial para testar e depurar o compilador. É sempre importante certificar-se de que o arquivo "teste1.j--" contenha um código-fonte válido na linguagem que está compilando.

Parametro.hs:

O arquivo `Parametro.hs` contém definições e funções relacionadas à análise de parâmetros e declarações na linguagem que o compilador suporta.

1. **Importações e Módulo**:
    - O módulo `Parametro` importa módulos necessários da biblioteca Parsec para análise sintática.
    - Também importa outros módulos relacionados ao projeto, como `DataTypes`, `Lex`, `Relacional` e `Tabelas`.
2. **Análise de Parâmetros**:
    - `parametro`: Esta função define a análise de um parâmetro. Ela analisa um tipo de dado (`t`) seguido por um identificador (`i`) e retorna uma

representação do parâmetro, onde o identificador está associado ao tipo.

- `parametros`: Essa função analisa uma lista de parâmetros usando a função `parametro`.

3. **Declarações**:

- `declaration`: Define a análise de declarações de variáveis. Ela analisa um tipo de dado (`t`) seguido por uma lista de identificadores (`i`) separados por vírgulas e terminados com um ponto e vírgula. A função retorna uma lista de pares onde cada identificador é associado ao tipo.

Essas definições são importantes para analisar a declaração de variáveis e parâmetros na linguagem. Elas são usadas para entender e representar as declarações de variáveis dentro do código-fonte.

Programa.hs:

O arquivo `Programa.hs` contém definições e funções relacionadas à análise de funções e programas na linguagem que o compilador suporta.

1. **Importações e Módulo**:

- O módulo `Programa` importa módulos necessários da biblioteca Parsec para análise sintática.
- Também importa outros módulos relacionados ao projeto, como `DataTypes`, `Lex`, `Relacional`, `Tabelas`, `Logico`, `Comandos` e `Parametro`.

2. **Análise do Programa**:

- `program`: Essa função define a análise do programa como um todo. Ela analisa a lista de funções definidas (`funcoes`) e o bloco principal do programa (`blocoPrincipal`). Em seguida, retorna um valor do tipo `Programa`, que representa o programa completo.

3. **Funções**:

- `funcao`: Essa função analisa a definição de uma função na linguagem. Ela começa com a análise do tipo de retorno (`tipoRetorno`), seguido pelo nome da função (`id`), uma lista de parâmetros (`p`), e o bloco de código da função (`bloco`). Retorna um par em que a primeira parte é um valor do tipo `Funcao` e a segunda parte é uma tupla contendo o nome da função, as declarações de variáveis locais (concatenadas a partir do bloco) e os comandos do bloco.

- `funcoes`: Essa função analisa várias definições de funções, retornando uma lista de funções. Ela utiliza a função `many` para analisar várias instâncias de `funcao`.

4. **Tipo de Retorno**:
   - `tipoRetorno`: Essa função analisa o tipo de retorno de uma função. Ela tenta analisar um tipo (como `TDouble`, `TInt`, `TString`, ou `TVoid`) ou a palavra-chave `void`. Isso é usado para determinar o tipo de retorno de uma função.

Essas definições são essenciais para analisar as funções e a estrutura geral do programa na linguagem. Elas são usadas para entender e representar a estrutura das funções e a composição do programa.

Relacional.hs:

O arquivo `Relacional.hs` contém definições e funções relacionadas à análise de operadores relacionais na linguagem que o compilador suporta.

1. **Importações e Módulo**:
   - O módulo `Relacional` importa módulos necessários da biblioteca Parsec para análise sintática.
   - Também importa outros módulos relacionados ao projeto, como `DataTypes` e `Lex`.

2. **Análise de Operadores Relacionais**:
   - `opr`: Esta função define a análise de operadores relacionais. Ela utiliza a função `operator` para analisar os operadores relacionais, como `==`, `>=`, `<=`, `>`, `<` e `/=`. Cada operador é associado a um construtor de tipo correspondente, como `(:==:)` para `==`, `(:>=:)` para `>=`, e assim por diante. Esses construtores são usados para representar as operações relacionais na árvore sintática.

3. **Listas de Elementos**:
   - `list element`: Esta função é definida para analisar listas de elementos separados por comandos (vírgulas ou ponto e vírgula). Ela utiliza a função `sepBy` para analisar listas de elementos separados por `element`.

Essas definições são importantes para a análise de operadores relacionais na linguagem. Os operadores relacionais são parte fundamental da análise de expressões lógicas e relacionais.

Tabelas.hs:

O arquivo `Tabelas.hs` contém definições e funções relacionadas à análise de operadores e tabelas de precedência na linguagem que o compilador suporta.

1. **Importações e Módulo**:
   - O módulo `Tabelas` importa módulos necessários da biblioteca Parsec para análise sintática.
   - Também importa outros módulos relacionados ao projeto, como `DataTypes` e `Lex`.
2. **Tabelas de Precedência**:
   - `tabela`: Esta definição cria uma tabela de precedência para operadores aritméticos, como adição, subtração, multiplicação e divisão. A tabela é uma lista de listas, onde cada lista interna contém operadores com a mesma precedência. Por exemplo, na primeira lista, temos o operador unário "-" associado ao construtor `Neg`, na segunda lista, temos operadores de multiplicação e divisão associados aos construtores `(:*:)` e `(:/::)`, respectivamente, e na terceira lista, temos operadores de adição e subtração associados aos construtores `(:+:)` e `(:-:)`, respectivamente.
   - `tabelaL`: Define uma tabela de precedência para operadores lógicos, como negação, conjunção e disjunção. Essa tabela é semelhante à `tabela` e é usada para lidar com operadores lógicos.
3. **Tipos de Dados**:
   - `tipo`: Essa função é responsável por analisar tipos de dados. Ela tenta analisar palavras-chave como "int", "double" ou "string" e retorna o tipo correspondente da linguagem, como `TInt` para "int", `TDouble` para "double" e `TString` para "string".

Essas definições são essenciais para a análise de expressões aritméticas e lógicas na linguagem. As tabelas de precedência definem como os operadores são agrupados em expressões, enquanto a função `tipo` é usada para determinar o tipo de dados em declarações e expressões.

Makefile:

O arquivo `Makefile` é um arquivo de script que contém instruções para compilar e executar o programa Haskell.

1. **Regras Make**:

- O Makefile contém três regras principais: `all`, `compile`, `run`, e uma regra adicional chamada `clean`. As regras são definidas no formato `nome_da_regra: comandos`.

2. `all`:
   - A regra `all` é a regra padrão que será executada quando você simplesmente chamar `make` no terminal. Ela depende das regras `compile` e `run`, o que significa que, quando você chamar `make`, ele primeiro compilará o código com a regra `compile` e, em seguida, executará o programa com a regra `run`.
3. `compile`:
   - A regra `compile` é responsável por compilar o código Haskell. Ela usa o compilador GHC (Glasgow Haskell Compiler) para compilar o código contido no arquivo `Main.hs`.
4. `run`:
   - A regra `run` é responsável por executar o programa compilado. Ela simplesmente chama o executável `Main` que foi gerado após a compilação.
5. `clean`:
   - A regra `clean` é usada para limpar arquivos temporários gerados durante a compilação. Ela remove arquivos com extensões `.o` (objetos), `.hi` (interfaces) e o executável `Main`.

Essas regras tornam mais conveniente o processo de compilação e execução do programa Haskell. Para usar o Makefile, basta abrir um terminal, navegar até o diretório onde seu código está localizado e executar `make` ou `make all` para compilar e executar o programa.

Lembrando que a regra `all` é a regra padrão, então você pode apenas digitar `make` e o Makefile executará `all`, que compila e executa o programa em sequência. Se desejar limpar arquivos temporários gerados durante a compilação, pode executar `make clean`.

Certifique-se de ter o GHC (Glasgow Haskell Compiler) instalado em seu sistema para que o Makefile funcione corretamente.

Teste1.j—:

O arquivo `teste1.j--` contém um exemplo de código-fonte escrito em nossa linguagem de programação.

1. **Definição de Funções**:
    - O código começa com a definição de três funções: `maior`, `fat`, e `imprimir`.
    - `maior` é uma função que recebe dois parâmetros do tipo `double` e retorna um valor do tipo `double`. Ela compara os dois parâmetros e retorna o maior deles.
    - `fat` é uma função que recebe um parâmetro do tipo `int` e retorna um valor do tipo `int`. Essa função calcula o fatorial do número recebido como parâmetro.
    - `imprimir` é uma função que recebe uma string (`s`) e um valor do tipo `double` (`r`). Ela imprime a string e o valor do tipo `double`. A função não tem um valor de retorno, pois tem o tipo `void`.
2. **Bloco Principal**:
    - Após as definições das funções, há um bloco principal sem nome que não recebe parâmetros. Este é o ponto de entrada do programa.
    - No bloco principal, são declaradas variáveis do tipo `int`, `double` e `string`.
    - O programa solicita a entrada de um número, calcula seu fatorial usando a função `fat`, chama a função `maior` para encontrar o maior valor entre 2.5 e 10, chama a função `imprimir` para imprimir uma mensagem e um número inteiro, e finalmente retorna 0.
3. **Condicionais e Loops**:
    - No código, é utilizado estruturas condicionais `if` e `else` para tomar decisões com base em condições. Por exemplo, no bloco principal, há uma condição que verifica se `a > b` e escolhe o maior valor.
    - Há também um loop `while` no código que é usado na função `fat` para calcular o fatorial de um número.
4. **Funções e Tipos de Dados**:
    - O código demonstra o uso de funções com diferentes tipos de parâmetros e retornos, como `double`, `int` e `void`.
    - Os tipos de dados usados no código incluem `double`, `int`, `string`, e `void`, que são parte da linguagem.

Este código de exemplo ilustra várias características comuns de uma linguagem de programação, como funções, condicionais, loops, tipos de dados e declarações de

variáveis. Ele esta relacionado a cálculos matemáticos, como cálculo do fatorial e comparação de números.

Conclusão

Neste projeto, implementamos a análise sintática para um compilador em Haskell, tornando a implementação uma tarefa mais acessível. Nossa principal ferramenta foi a biblioteca Parsec, e optamos por estruturar o código em diversos módulos para manter a organização. Na segunda fase do trabalho, será feita a análise semântica, que envolverá a verificação de tipos, já na terceira fase, iremos desenvolver a geração de código a partir da representação intermediária, permitindo que programas escritos em nossa linguagem sejam executados em máquinas reais.