

## Definição e implementação da linguagem *Trivia*

Departamento de Computação  
 Universidade Federal de Ouro Preto  
 Prof. José Romildo Malaquias  
 16 de junho de 2021

### Resumo

*Trivia* é uma pequena linguagem de programação usada para fins didáticos na aprendizagem de técnicas de construção de compiladores.

Na disciplina de construção de compiladores serão propostas atividades de documentação e implementação de *Trivia*.

## Sumário

<b>1 A linguagem <i>Trivia</i></b>	<b>1</b>
<b>2 O projeto</b>	<b>4</b>
2.1 Estrutura do projeto . . . . .	4
2.2 Módulos importantes . . . . .	5
2.3 Implementando uma nova construção de <i>Trivia</i> . . . . .	6
<b>3 Mensagens de erro</b>	<b>7</b>
<b>4 Símbolos</b>	<b>7</b>
<b>5 O analisador léxico</b>	<b>8</b>

## 1 A linguagem *Trivia*

No livro *Introduction to Compiler Design*, Torben apresenta uma pequena linguagem de programação para fins didáticos. Nesta disciplina vamos considerar uma linguagem semelhante, que chamaremos de *Trivia*, baseada na linguagem apresentada no livro, porém com algumas diferenças sintáticas e semânticas.



*Trivia* é uma linguagem de programação bastante simples que será usada para a prática de técnicas usadas na implementação de compiladores.

As construções da linguagem serão detalhadas no decorrer do curso. Começaremos com uma versão básica, e oportunamente serão apresentadas versões mais aprimoradas, com novas construções.

*Trivia* é uma **linguagem funcional de primeira ordem** com definições recursivas e tipagem estática. A sintaxe é apresentada na gramática 1.

$Program \rightarrow Funs$	programa
$Funs \rightarrow Fun$	
$Funs \rightarrow Fun \ Funs$	
$Fun \rightarrow TypeId \ ( \ TypeIds \ ) \ = \ Exp$	<b>declaração de função</b>
$TypeId \rightarrow int \ id$	<b>tipo e identificador</b>
$TypeId \rightarrow bool \ id$	
$TypeIds \rightarrow TypeId$	<b>lista de parâmetros</b>
$TypeIds \rightarrow TypeId \ , \ TypeIds$	
$Exp \rightarrow num$	<b>literal inteiro</b>
$Exp \rightarrow id$	<b>variável</b>
$Exp \rightarrow Exp \ + \ Exp$	<b>operações aritméticas</b>
$Exp \rightarrow Exp \ < \ Exp$	<b>operações relacionais</b>
$Exp \rightarrow if \ Exp \ then \ Exp \ else \ Exp$	<b>expressão condicional</b>
$Exp \rightarrow id \ ( \ Exps \ )$	<b>chamada de função</b>
$Exp \rightarrow let \ id \ = \ Exp \ in \ Exp$	<b>expressão de declaração</b>
$Exps \rightarrow Exp$	<b>lista de argumentos</b>
$Exps \rightarrow Exp \ , \ Exps$	

Gramática 1: Linguagem Trivia

Esta gramática é claramente **ambígua**, como ilustra as figuras 1, 2, e 3. As ambiguidades podem ser resolvidas observando

- uma definição de **precedência** relativa e **associatividade** para os operadores, indicadas pela tabela 1, em ordem decrescente de precedência, e
- que as expressões condicionais e de declaração de variável se estendem ao máximo para a direita.

operadores	associatividade
+	esquerda
<	-

Tabela 1: Precedência e associatividade dos operadores

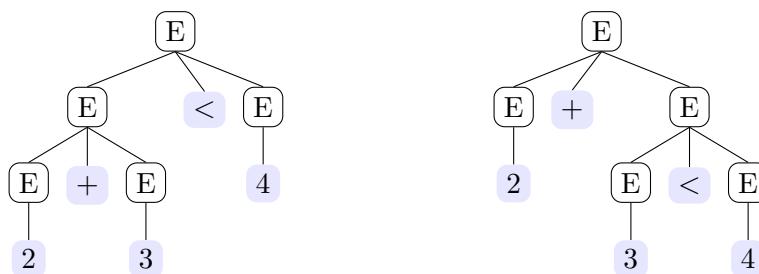


Figura 1: Árvores sintáticas para a cadeia  $2 + 3 < 4$

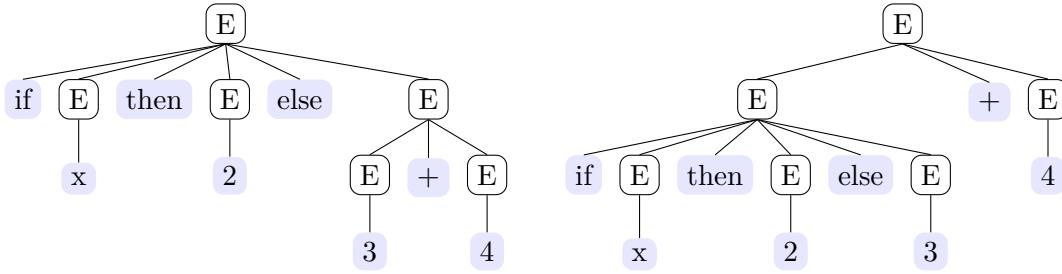


Figura 2: Árvores sintáticas para a cadeia `if x then 2 else 3 + 4`

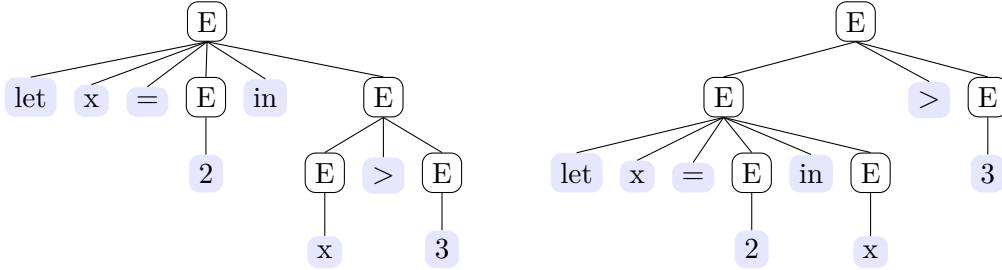


Figura 3: Árvores sintáticas para a cadeia `let x = 2 in x < 3`

Um **programa** em *Trivia* é uma lista de declarações de **funções**. As funções são **mutuamente recursivas**, e nenhuma função pode ser declarada mais de uma vez. Cada função declara o tipo do seu resultado e o tipo e o nome de seus parâmetros formais. Não pode haver repetição de nome na lista de parâmetros de uma função. Funções e variáveis têm espaços de nome separados. O corpo de uma função é uma **expressão**.

Os **tipos** de *Trivia* são `int` (inteiro) e `bool` (booleano).

Uma **expressão** pode ser:

- uma constante inteira
- uma variável
- uma operação aritmética
- uma operação relacional
- uma expressão condicional
- uma chamada de função
- uma expressão com uma declaração local

Comparações são definidas para inteiros e para booleanos (sendo *false* considerado menor que *true*). Mas operações aritméticas são definidas apenas para inteiros.

Um programa deve conter uma função chamada `main`, com um parâmetro inteiro e que retorna um inteiro. Um programa é executado pela chamada desta função com um argumento, que deve ser inteiro. O resultado desta função é a saída do programa.

Ocorrências de **caracteres brancos** (espaço, tabulação horizontal, nova linha) entre os símbolos léxicos são ignorados.

Os **literais inteiros** são formados por uma sequência de um ou mais dígitos decimais. Não há literais inteiros negativos. São exemplos de literais inteiros:

```
2014  
872834  
0  
0932
```

**Identificadores** são sequências de letras maiúsculas ou minúsculas, dígitos decimais e sublinhados (\_), começando com uma letra. Letras maiúsculas e minúsculas são distintas em um identificador. Identificadores são usados para nomear entidades usadas em um programa, como funções e variáveis. São exemplos de identificadores:

```
peso  
idadeAluno  
alfa34  
primeiro_nome
```

Não são exemplos de identificadores:

```
--peso  
idade do aluno  
34rua  
primeiro-nome  
x'
```

**Palavras-chave**, usadas em algumas construções sintáticas da linguagem, são reservadas, isto é, não podem ser usadas como identificadores.

Exemplo de programa em *Trivia*:

```
int f(int a, int b) =  
    let y = a + b  
    in  
        if a < b then  
            f(sucessor(a), b) + y  
        else  
            y + y  
  
int sucessor(int n) =  
    n + 1  
  
int main(int x) =  
    f(x, 10)
```

## 2 O projeto

### 2.1 Estrutura do projeto

O projeto será desenvolvido na linguagem OCaml usando a ferramenta dune para automatização da compilação. O projeto usa algumas ferramentas e bibliotecas externas, entre as quais citamos:

**ocamllex** Ocamllex é um gerador de analisador léxico. Ele produz analisadores léxicos (em

OCaml) a partir de conjuntos de expressões regulares com ações semânticas associadas. É distribuído junto com o compilador de OCaml.

**menhir** Menhir é um gerador de analisador sintático. Ele transforma especificações gramaticais de alto nível, decoradas com ações semânticas expressas na linguagem de programação OCaml, em analisadores sintáticos, também expressos em OCaml.

**dune** Dune é um sistema de compilação para OCaml.

**ppx\_import** É uma extensão de sintaxe que permite extrair tipos ou assinaturas de outros arquivos de interface compilados

**ppx\_deriving** É uma extensão de sintaxe que facilita geração de código baseada em tipos, em OCaml

**ppx\_expect** É uma extensão de sintaxe para escrita de testes inline de expectativa, em OCaml

**camomile** Camomile é uma biblioteca unicode para OCaml.

O código é organizado segundo a estrutura de diretórios mostrada na figura 4.

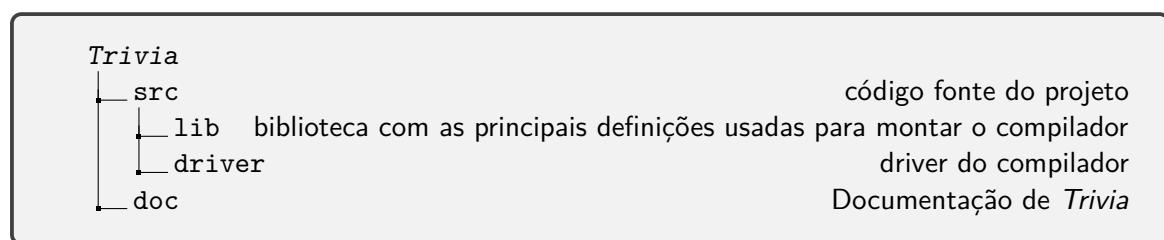


Figura 4: Estrutura de diretórios do projeto do compilador

Existem outros diretórios gerados automaticamente que não são relevantes nesta discussão e por isto não foram mencionados. Caso algum ambiente de desenvolvimento integrado (IDE) seja usado, provavelmente haverá alguns arquivos e diretórios específicos do ambiente e que também não são mencionados.

Os arquivos `src/lib/dune` e `src/driver/dune` contém as especificações do projeto esperadas pelo dune. Neles são indicadas informações como nome do projeto, dependências externas e flags necessários para a compilação da biblioteca e da aplicação.

## 2.2 Módulos importantes

Alguns módulos importantes no projeto são mencionados a seguir. Alguns já estão prontos, e outros deverão ser criados ou modificados pelo participante nas atividades do curso.

- O módulo `Absyn` contém os tipos que representam as árvores sintáticas abstratas para as construções da linguagem fonte.
- O módulo `Absyntree` contém funções para converter árvores de sintaxe abstratas para árvores genéricas de strings, úteis para visualização das árvores sintáticas durante o desenvolvimento do compilador.
- O módulo `Lexer` contém as declarações relacionadas com o analisador léxico do compilador.

O analisador léxico (módulo `Lexer`) é gerado automaticamente pelo `ocamllex`. A especificação léxica é feita no arquivo `src/lib/lexer.mll` usando expressões regulares.

- O módulo `Parser` contém as declarações relacioandas com o analisador sintático do compilador.

O analisador sintático (módulo `Parser`) é gerado automaticamente pelo `menhir`. A especificação sintática é feita no arquivo `src/lib/parser.mly` usando uma gramática livre de contexto.

- O módulo `Semantic` contém declarações usadas na análise semântica e geração de código do compilador.
- O módulo `Symbol` contém declarações que implementam um tipo usado para representar nomes de identificadores de forma eficiente, e será discutido posteriormente.
- O módulo `Env` contém declarações para a manipulação dos ambientes de compilação (às vezes também chamados de contexto). Estes ambientes são representados usando tabelas de símbolos.
- O módulo `Types` contém declarações para a representação interna dos tipos da linguagem fonte.
- O módulo `Error` contém declarações usadas para reportar errors detectados pelo compilador durante a compilação.
- O módulo `Location` contém declarações usadas para representação de localizações de erros no código fonte, importantes quando os erros forem reportados.
- O módulo `Driver` é formado por declarações, incluindo a função `main`, ponto de entrada para a execução do compilador.

### 2.3 Implementando uma nova construção de *Trivia*

Ao acrescentar uma nova construção na implementação da linguagem, procure seguir os seguintes passos:

- Se necessário defina um novo construtor de dados no tipo `Types.t` para representar algum tipo da linguagem *Trivia* que ainda não faça parte do projeto. *[Análise semântica]*
- Se necessário acrecente ao ambiente inicial (no módulo `Env`) a representação de quaisquer novos tipos, variáveis ou funções que façam parte da biblioteca padrão de *Trivia*. *[Análise semântica]*
- Defina os novos construtores de dados necessárias para representar a árvore abstrata para a construção no tipo `Absyn.t`.
  - definir os campos necessários para as sub-árvores da árvore abstrata, se houver
  - extender as funções do `Absyntree` para permitir a conversão para uma árvore de strings, útil para visualização gráfica da árvore abstrata.

*[Análise sintática]*

- Extenda a função de análise semântica `Semantic.semantic` para verificar a nova construção. *[Análise semântica]*

- Declare quaisquer novos símbolos terminais e não-terminais na gramática livre de contexto da linguagem que se fizerem necessários para as especificações léxica e sintática da construção. *[Análises léxica e sintática]*
- Acrescente as regras de produção para a construção na gramática livre de contexto da linguagem, tomando o cuidado de escrever ações semânticas adequadas para a construção da árvore abstrata correspondente. Se necessário use declarações de precedência de operadores. *[Análise sintática]*
- Se necessário acrescente regras léxicas que permitam reconhecer os novos símbolos terminais da especificação léxica da linguagem. *[Análise léxica]*

## 3 Mensagens de erro

O projeto contém algumas funções para **reportar erros** encontrados durante a compilação. Estas funções fazem parte do módulo `Error` e serão comentadas a seguir.

Em todo compilador é desejável que os erros encontrados sejam reportados com uma indicação da **localização** do erro, acompanhada por uma **mensagem** explicativa do problema ocorrido. Para tanto torna-se necessário guardar a informação da localização em que cada frase do programa (o que corresponde a cada nó da árvore abstrata construída para representar o programa) foi encontrada. O módulo `Location` contém algumas definições relacionadas com estas localizações.

Neste projeto as localizações no código fonte são representadas pelo tipo `Location.t`, que leva em consideração as posições no código fonte onde a frase começou e terminou.

Cada uma destas posições é do tipo `Lexing.position`. O módulo `Lexing` faz parte da biblioteca padrão do OCaml e será extensivamente usado nas implementações dos analisadores léxico e sintático. O tipo `Lexing.position` contém as seguintes informações:

- a indicação da unidade (arquivo fonte) sendo compilada,
- o número da linha, e
- o número da coluna

A função `Error.error`, e outras funções similares encontradas no módulo `Error`, devem ser usadas para emissão de mensagens de erro. Esta função recebe como argumentos a localização do erro, a mensagem de formatação de diagnóstico, e possivelmente argumentos complementares de acordo com a mensagem de formatação.

## 4 Símbolos

Linguagens de programação usam **identificadores** para nomear entidades da linguagem, como tipos, variáveis, funções, classes, módulos, etc.

**Símbolos léxicos** (também chamados de **símbolos terminais** ou **tokens**) que são classificados como identificadores têm um valor semântico (atributo) que é o nome do identificador. A princípio o valor semântico de um identificador pode ser representado por uma cadeia de caracteres (tipo `string` do OCaml). Porém o tipo `string` tem algumas inconveniências para o compilador:

- Geralmente o mesmo identificador ocorre várias vezes em um programa. Se cada ocorrência for representada por uma string (ou seja, por uma sequência de caracteres), o uso de memória poderá ser grande.
- Normalmente existem dois tipos de ocorrência de identificadores em um programa:

- uma declaração do identificador, e
- um ou mais usos do identificador já declarado.

Durante a compilação cada ocorrência de uso de um identificador deve ser associada com uma ocorrência de declaração. Para tanto os identificadores devem ser comparados para determinar se são iguais (isto é, se tem o mesmo nome). O uso de strings é ineficiente, pois pode ser necessário comparar todos os caracteres das strings para determinar se elas são iguais ou não.

Por estas razões o compilador utiliza o tipo `Symbol.symbol` para representar os nomes dos identificadores. Basicamente usa-se uma tabela *hash* onde os identificadores são colocados à medida que eles são encontrados. Sempre que o analisador léxico encontrar um identificador, deve-se verificar se o seu nome já está na tabela. Em caso afirmativo, usa-se o símbolo correspondente que já se encontra na tabela. Caso contrário cria-se um novo símbolo, que é adicionado à tabela associado ao nome encontrado, e este novo símbolo é usado pelo analisador léxico como valor semântico do *token*.

Na implementação de `Symbol.symbol` associa-se a cada novo símbolo um número inteiro diferente. A comparação de igualdade de símbolos se resume a uma comparação (muito eficiente) de inteiros, já que o mesmo identificador estará sempre sendo representado pelo mesmo símbolo (associado portanto ao mesmo número inteiro).

A função `Symbol.symbol` cria um símbolo a partir de uma string, e a função `Symbol.name` obtém o nome de um símbolo dado.

## 5 O analisador léxico

O módulo `Lexer` contém as declarações que implementam o analisador léxico do compilador. Este módulo será gerado automaticamente pela ferramenta `ocamllex`.

A especificação da estrutura léxica da linguagem fonte é feita no arquivo `src/lib/lexer.mll` usando **expressões regulares**. Consulte a documentação do `ocamllex` para entender como fazer a especificação léxica.

Os analisadores léxico e sintático vão se comunicar durante a compilação, pois os tokens obtidos pelo analisador léxico serão consumidos pelo analisador sintático. Ou seja, os tokens são os símbolos terminais da gramática usada pelo gerador de analisador sintático. Para manter a consistência dos analisadores léxico e sintático os *símbolos terminais (tokens)* são declarados na gramática livre de contexto do `menhir`, no arquivo `src/lib/parser.mly`. Consulte a documentação do `menhir` para entender como escrever a gramática livre de contexto.