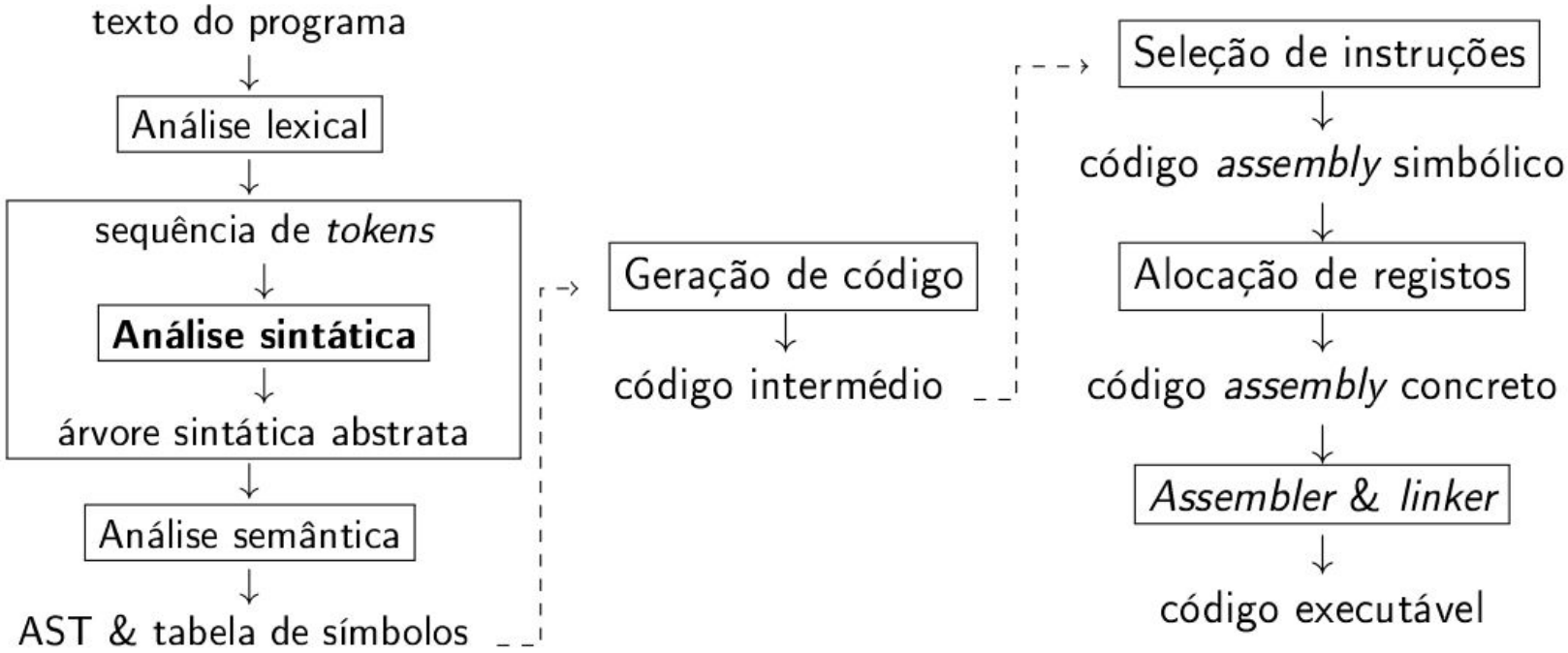
Linguagens de Programação

Geradores de Parsers

Samuel da Silva Feitosa

Aula 17

Relembrando: fases de um compilador



Geradores de Analisadores Sintáticos

- Um parser pode ser gerado automaticamente.
  - Yacc: “Yet Another Compiler Compiler”, gerador de analisadores sintáticos do sistema UNIX (para a linguagem C).
  - Bison: uma reimplementação GNU do Yacc (para C ou C++).
  - Happy: gerador semelhante ao Yacc e Bison que produz analisadores sintáticos para Haskell.
- Existem diversos outros, tanto para Haskell, como para outras linguagens.
- Vamos desenvolver um *parser* para uma das gramáticas simples que vimos na aula anterior.

Primeiro passo: Análise Léxica

- É comum que seja escrita uma função simples para realizar a análise léxica do código fonte em Haskell.
- Para isso, vamos criar um arquivo chamado Lexer.hs.
  - Nele vamos definir os *tokens* válidos.
  - E implementar uma função que recebe uma string contendo o código-fonte e retorna uma lista de *tokens*.

Exemplo: Lexer.hs (1)

- Definição dos tokens válidos:

```haskell
module Lexer where

import Data.Char

-- Lista os tokens válidos
data Token = TokenNum Int
           | TokenPlus
           | TokenTimes
           | TokenLParen
           | TokenRParen
           deriving Show
```

Exemplo: Lexer.hs (2)

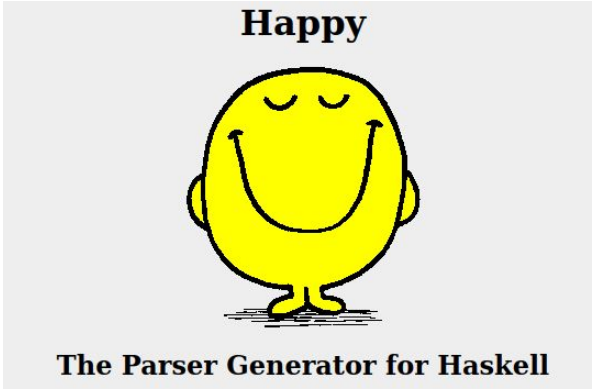
- Função que realiza a análise léxica:

```haskell
-- Função que recebe o código e retorna uma lista de tokens
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
    | isSpace c = lexer cs
    | isDigit c = lexNum (c:cs)
lexer ('+':cs) = TokenPlus : lexer cs
lexer ('*':cs) = TokenTimes : lexer cs
lexer ('(':cs) = TokenLParen : lexer cs
lexer (')':cs) = TokenRParen : lexer cs
lexer _ = error "Lexical error: caracter inválido!"

lexNum cs = case span isDigit cs of
                (num, rest) -> TokenNum (read num) : lexer rest
```

Segundo passo: Análise Sintática

- **Happy** é um gerador de analisadores sintáticos para Haskell.
- Recebe um arquivo com a **gramática da linguagem anotada**.
  - Produções anotadas com **ações semânticas** (expressões Haskell).
- Gera automaticamente o código do analisador sintático.
  - Vamos criar o arquivo Parser.y e executar:
  - Parser.y → Happy → Parser.hs
  - `$ happy Parser.y`
- Documentação:
  - https://www.haskell.org/happy/



Exemplo: Parser.y (1)

- Diretiva %name define o nome da função de parsing.
- %tokentype define o tipo de dado que representa os Tokens (definido no arquivo Lexer.hs).
- %error indica qual é a função a ser chamada em caso de erro no processamento da gramática.
- %token define qual construtor utilizar a partir da sintaxe concreta da linguagem.

```
1   {
2   module Parser where
3
4   import Lexer
5   }
6
7   %name parser
8   %tokentype { Token }
9   %error { parseError }
10
11  %token
12      num     { TokenNum $$ }
13      '+'     { TokenPlus }
14      '*'     { TokenTimes }
15      '('     { TokenLParen }
16      ')'     { TokenRParen }
```

Exemplo: Parser.y (2)

- Não esquecer de colocar o %% da linha 18.
- **Exp** define um **não-terminal** para expressões (mesmo exemplo que vimos na aula anterior).
  - Ações semânticas são colocadas a direita, entre parênteses (ainda não realizado).
- **parseError** é a função que define o que fazer em caso de erro.
- Neste exemplo todas as produções retornam ().
  - Ou seja, nenhum resultado útil está sendo produzido ainda.
- Ao finalizar, rodar **happy Parser.y**.
  - Será gerado o arquivo Parser.hs

```
18  %%
19
20  Exp     : num           { () }
21          | Exp '+' Exp   { () }
22          | Exp '*' Exp   { () }
23          | '(' Exp ')'   { () }
24
25  {
26
27  parseError :: [Token] -> a
28  parseError _ = error "Syntax error!"
29
30  }
```

Exemplo: Main.hs

- Este programa lê toda a entrada padrão e verifica se respeita a gramática.
  - Se sim, imprime ().
  - Caso contrário, lança uma exceção.

```
1  module Main where
2
3  import Lexer
4  import Parser
5
6  main = getContents >>= print . parser . lexer
```

Representação Abstrata

- Além de reconhecer uma linguagem um compilador/interpretador deve construir uma **representação do programa**.
  - Essa representação geralmente é feita através de uma Árvore de Sintaxe Abstrata (AST).
- O Happy facilita o processo de criação de uma AST a partir da sintaxe concreta da linguagem.

Construir a árvore sintática: Parser.y

- Foi declarado um novo tipo algébrico para representar a árvore sintática (linhas 27-31).
- Foi acrescentada uma ação a cada produção que constrói uma árvore a partir das sub-árvores:
  - $1, $2, etc. referem-se aos valores semânticos dos terminais e não-terminais.
  - Os valores de não-terminais são árvores Exp.
  - Os valores de terminais são definidos na seção %token. Ex.: num { TokenNum $$ } (o valor de TokenNum 10 é o inteiro 10).

```
18  %%
19
20  Exp     : num           { Num $1 }
21          | Exp '+' Exp   { Add $1 $3 }
22          | Exp '*' Exp   { Times $1 $3 }
23          | '(' Exp ')'   { Paren $2 }
24
25  {
26
27  data Expr = Num Int
28            | Add Expr Expr
29            | Times Expr Expr
30            | Paren Expr
31            deriving Show
32
33  parseError :: [Token] -> a
34  parseError _ = error "Syntax error!"
35
36  }
```

Conflitos (1)

- O Happy reporta **conflitos** se a gramática for ambígua.

Parser.y

```
⋮
Exp : num                    { ... }
    | Exp '+' Exp            { ... }
    | Exp '*' Exp            { ... }
⋮
```

```
$ happy Parser.y
shift/reduce conflicts:  4
```

Conflitos (1)

- Por omissão, os conflitos são automaticamente resolvidos.
  - Porém, dependendo da gramática isso pode não ser correto.
- Algumas vezes é necessário reescrever a gramática para remover ambiguidades.
- Como alternativa, pode-se especificar a **associatividade** e **precedência** de *tokens*.
  - Associatividades podem ser: %left, %right, %nonassoc

```
%nonassoc '<' '>' '=='          precedência mais baixa
%left '+' '-'
%left '*' '/'                   precedência mais elevada
```

Conflitos (1)

- A associatividade permite resolver **ambiguidades que envolvem um só operador**.
  - Exemplo: 1 + 2 + 3.
  - Usando %left ‘+’, a expressão é analisada como (1 + 2) + 3.
  - Usando %right ‘+’, a expressão é analisada como 1 + (2 + 3).
- A ordem de precedência permite resolver **ambiguidades entre operadores**.
  - Exemplo: 1 + 2 * 3.
  - Como a precedência de * é maior que a de +, a expressão 1 + 2 * 3 é avaliada como 1 + (2 * 3).

Considerações Finais

- Nesta aula estudamos como utilizar o Happy como um gerador de analisador sintático.
  - Vimos como escrever uma função que realiza a análise léxica, ou seja, a partir de uma string, gera uma sequência de *tokens*.
  - Vimos como criar uma árvore de sintaxe abstrata, a qual pode ser utilizada para realizar a interpretação das instruções da linguagem.