Geradores de analisadores LALR

Construção de compiladores I

Objetivos

Objetivos

- Apresentar o gerador de analisadores sintáticos LALR para Haskell.

Introdução

Introdução

- Nas aulas anteriores, vimos alguns algoritmos LR.
  - Em especial, o LALR.
- Este algoritmo consegue realizar o parsing de construções de linguagens de programação.

Introdução

- Nesta aula, veremos uma ferramenta para produzir estes analisadores automaticamente.
- Este tipo de ferramenta é disponível em praticamente todas as linguagens de programação.

Introdução

- Em Haskell, o analisador que utilizaremos é o Happy, capaz de produzir analisadores LALR a partir de gramáticas.

Introdução

- Apresentaremos a linguagem do Happy, utilizando uma pequena linguagem de expressões.

Especificações Happy

Especificações Happy

- Especificações são formadas por:
  - Trechos de código Haskell
  - Declarações de tokens, erro e nome do parser
  - Definição da gramática.

Especificações Happy

- Trechos de código Haskell são usados para:
  - Definir funções auxiliares.
  - Definir tipos de dados a serem usados pelo parser.

Especificações Happy

- Declarações

```
 expParser
%tokentype { Token }
%error { parseError }
```

Especificações Happy

- Definições de tokens

```
%token
     int  {TNumber $$}
     var  {TVar $$}
     '+'  {TAdd}
     '*'  {TMul}
     '('  {TLParen}
     ')'  {TRParen}
```

Especificações Happy

- Definição da gramática

```
Expr   : Term '+' Expr   {Add $1 $3}
       | Term            {$1}

Term   : Factor '*' Term {Mul $1 $3}
       | Factor          {$1}

Factor : int             {Number $1}
       | var             {Var $1}
       | '(' Expr ')'    {$2}
```

Especificações Happy

- Podemos produzir um parser LALR a partir da especificação mostrada.

```
happy Parser.y -o Parser.hs
```

Especificações Happy

- Produzindo o conjunto de itens
  - Produzindo arquivo Parser.info

```
happy -i Parser.y
```

Especificações Happy

- Exemplo de gramática com conflitos de shift/reduce
- Solucionando o conflito com predências.

Mônadas

Mônadas

- Para construir parsers com suporte
  - Manter contagem da linha atual.
  - Melhor diagnóstico de erros.
- Precisamos de mônadas

Mônadas

- Tanto o Happy quanto o Alex possuem suporte a mônadas para lidar com estado (número de linhas e colunas).
- Vamos iniciar com mudanças na especificação Alex.

Mônadas

- Inserimos suporte a mônadas em especificações Alex usando o wrapper monadUserState
- Com isso, ações para produzir tokens passam a ter o tipo
  - Tipo Alex: mônada usada pelo analisador léxico.

```
type Action a = AlexInput -> Int -> Alex a
```

Mônadas

- Tipo AlexInput

```
type AlexInput =
  ( AlexPosn      -- current position,
  , Char          -- previous char
  , [Byte]        -- rest of the bytes for the current char
  , String        -- current input string
  )
```

Mônadas

- Tipo AlexPosn

```
data AlexPosn = AlexPn
  !Int            -- absolute character offset
  !Int            -- line number
  !Int            -- column number
```

Mônadas

- Produzindo tokens

```
tokenInteger :: Action RangedToken
tokenInteger inp@(_, _, _, str) len
  = pure RangedToken {
           token_ = TNumber $ read $ take len str
         , range_ = mkRange inp len
         }
```

Mônadas

- Tipo RangedToken

```
data Range
  = Range {
      start_ :: AlexPosn
    , stop_  :: AlexPosn
    }
data RangedToken
  = RangedToken {
      token_ :: Token
    , range_ :: Range
    } deriving (Eq, Show)
```

Mônadas

- Com isso, resolvemos o problema de produzir informações de posições fazendo com que o analisador léxico retorne RangedToken's.

Mônadas

- Outro problema: como permitir comentários de várias linhas?
  - Comentários aninhados?

Mônadas

- Para lidar com isso, vamos criar um estado para armazenar o nível de comentários aninhados.
  - Tipo deve possuir esse nome

```
data AlexUserState
  = AlexUserState {
```

```
    level :: Int
  }
```

Mônadas

- Funções da mônada

```
get :: Alex AlexUserState
get = Alex $ \s -> Right (s, alex_ust s)

put :: AlexUserState -> Alex ()
put s' = Alex $ \s -> Right (s{alex_ust = s'}, ())

modify :: (AlexUserState -> AlexUserState) -> Alex ()
modify f = Alex $ \s -> Right (s{alex_ust = f (alex_ust s)}, ())
```

Mônadas

- Funções para lidar com aninhamento

```
nestComment :: Action RangedToken
nestComment input len = do
  modify $ \s -> s{level = level s + 1}
  skip input len
```

Mônadas

- Funções para lidar com aninhamento

```
unnestComment :: Action RangedToken
unnestComment input len = do
  state <- get
  let level' = level state - 1
  put state{level = level'}
  when (level' == 0) $
    alexSetStartCode 0
  skip input len
```

Mônadas

- Definição do estado inicial

```
alexInitUserState :: AlexUserState
alexInitUserState = AlexUserState 0
```

Mônadas

- Definição de tokens com marcações de estados

```
tokens :-
<0>  $white+                ; -- removing whitespace
<0>  @number                { tokenInteger }
<0>  @identifier            { tokenId }
<0>  \+                     { simpleToken TAdd }
...
<0>  "{-"                   { nestComment `andBegin` comment }
<0>  "-}"                   { \ _ _ -> alexError "Error: unexpected close comment!" }
```

Mônadas

- Definição de tokens com marcações de estados

```
<comment> "{-"              { nestComment }
<comment> "-}"              { unnestComment }
<comment> .                 ;
<comment> \n                ;
```

Mônadas

- Mudanças na especificação Happy.

Mônadas

- Definição de uma mônada

```
%monad { Alex }{ >>= }{ pure }
%lexer { lexer }{ RangedToken EOF _ }
```

Mônadas

- Especificação dos tokens

```
%token
     int  {RangedToken (TNumber _) _}
     var  {RangedToken (TVar _) _}
     '+'  {RangedToken TAdd _}
     '*'  {RangedToken TMul _}
     '('  {RangedToken TLParen _}
     ')'  {RangedToken TRParen _}
```

Mônadas

- Combinando posições de linha

```
(<->) :: Range -> Range -> Range
(Range a1 _) <-> (Range _ b2) = Range a1 b2
```

Mônadas

- Especificação da gramática

```
Expr   : Expr '+' Expr   {Add ((info $1) <-> (info $3)) $1 $3}
Expr   : Expr '*' Expr   {Mul ((info $1) <-> (info $3)) $1 $3}
       | int             {Number (info $1) (unNumber $1)}
       | var             {Var (info $1) (unId $1)}
       | '(' Expr ')'    {$2}
```

Concluindo

Concluindo

- Nesta aula apresentamos o gerador de analisadores sintáticos Happy.
- Mostramos como adicionar suporte a marcação de linhas / colunas.

Concluindo

- Próximas aulas: Semântica formal e interpretadores.

Exercícios

Exercícios

- Utilizando o gerador de analisadores sintáticos Happy, construa um analisador sintático para sintaxe de fórmulas da lógica proposicional.