Um Compilador Simples

Construção de compiladores I

Objetivos

Objetivos

- Apresentar a especificação léxica e sintática de uma linguagem simples.
- Apresentar a implementação completa de um compilador para uma linguagem simples de expressões.

Expressões Aritméticas

Expressões Aritméticas

- Especificação léxica
  - Números: digit+
  - Operadores e separadores: +, *, (,)

Expressões Aritméticas

- Especificação sintática

$$e \rightarrow n \mid e + e \mid e * e \mid (e)$$

Expressões Aritméticas

- De posse da gramática, vamos considerar as diferentes etapas do compilador.
  - Análise léxica
  - Análise sintática
  - Intepretador
  - Geração de código

Análise léxica

Análise léxica

- Primeira etapa do front-end de um compilador.
- Simplificar a entrada para análise sintática.

Análise léxica

- Simplificações:
  - Remoção de espaços em branco.
  - Remoção de comentários.
- Resultado: lista de **tokens**.

Análise léxica

- Token
  - Componente indivisível da sintaxe de uma linguagem.
- Exemplos:
  - identificadores
  - palavras reservadas
  - separadores
  - literais

Análise léxica

- Como implementar a análise léxica?

Análise léxica ad-hoc

- Percorra a string:
  - Se for um dígito, guarde-o para formar um número.
  - Se for um operador, gere o token.
  - Se for um parêntesis, gere o token.
  - Se for um espaço, tente gerar um número e descarte o espaço.

Análise léxica ad-hoc

- Como representar tokens?
  - Primeiro, definimos seus tipos: lexemas.

```
data Lexeme
  = TNumber Value
  | TPlus
  | TMul
  | TLParen
  | TRParen
```

Análise léxica ad-hoc

- Token = lexema + posição

```
data Token
  = Token {
      lexeme :: Lexeme
    , position :: (Int, Int)
    }
```

Análise léxica ad-hoc

- Configuração do analisador léxico
  - Linha e coluna atual.
  - String de dígitos consecutivos encontrados.
  - Lista de tokens encontrados.

```
type Line = Int
type Column = Int
type State = (Line, Column, String, [Token])
```

Análise léxica ad-hoc

- Transição de estado sob um caractere

```
transition :: State -> Char -> Either String State
transition state@(l, col, t, ts) c
  | c == '\n' = mkDigits state c
```

```
  | isSpace c = mkDigits state c
  | c == '+' = Right ( l, col + 1, ""
                     , mkToken state (Token TPlus (l,col)) ++ ts)
  | isDigit c = Right (l, col + 1, c : t, ts)
  | otherwise = unexpectedCharError l col c
```

Análise léxica ad-hoc

- Criando token de dígitos.

```
mkDigits :: State -> Char -> Either String State
mkDigits state@(l, col, s, ts) c
  | null s = Right state
  | all isDigit s
      = let t = Token (TNumber (VInt (read $ reverse s))) (l,col)
            l' = if c == '\n' then l + 1 else l
            col' = if c /= '\n' && isSpace c then col + 1 else col
        in Right (l', col', "", t : ts)
  | otherwise = unexpectedCharError l col c
```

Análise léxica ad-hoc

- Criando tokens

```
mkToken :: State -> Token -> [Token]
mkToken (l,c, s@(_ : _), _) t
  | all isDigit s = [t, Token (TNumber (VInt (read $ reverse s))) (l,c)]
  | otherwise = [t]
mkToken _ t = [t]
```

Analise léxica ad-hoc

- Analisador léxico
  - Caminhamento na entrada feito pela função foldl

```
lexer :: String -> Either String [Token]
lexer = either Left (Right . extract) . foldl step start
  where
    start = Right (1,1,"",[])
    step ac@(Left _) _ = ac
    step (Right state) c = transition state c
```

```
extract (l, col, s, ts)
  | null s = reverse ts
  | otherwise
    = let t = Token (TNumber (VInt (read $ reverse s))) (l, col)
      in reverse (t : ts)
```

Análise sintática

Análise sintática

- Vamos considerar uma técnica de análise sintática chamada de análise descendente recursiva.
- Permite a construção manual de analisadores sintáticos.

Análise sintática

- Analisador descendente recursivo
  - Uma função para cada não terminal da gramática

Análise sintática

- Lados direitos de regra como corpo da função
  - Caso um elemento de regra seja um token, consumimos este token
  - Caso seja um não terminal, chamamos a função correspondente.

Análise sintática

- Analisadores descentes recursivos não podem ser implementados para gramáticas recursivas à esquerda.
  - Gramáticas recursivas à esquerda geram parsers que entram em loop infinito!

Análise sintática

- Gramática para expressões
  - Regras recursivas à esquerda.

$$E \rightarrow n \mid E + E \mid E * E \mid (E)$$

Análise sintática

- Gramática para expressões
  - Sem regras recursivas à esquerda.

$$\begin{array}{lcl} E & \rightarrow & T + E \mid T \\ T & \rightarrow & F * T \mid F \\ F & \rightarrow & n \mid (E) \end{array}$$

Análise sintática

- De posse de uma gramática adequada, podemos partir para uma implementação.
- Em linguagens funcionais, analisadores descendentes recursivos são implementados como combinadores.

Análise sintática

- Um parser é uma função:
  - Entrada uma lista de tokens (tipo s)

Análise sintática

- Resultado: uma lista de pares de resultados e o restante de tokens.
  - Lista vazia: erro
  - Lista unitária: resultado único
  - Lista com mais de um resultado: possibilidade de backtracking.

Análise sintática

- Representando em Haskell

```
newtype Parser s a
  = Parser { runParser :: [s] -> [(a, [s])] }
```

Análise sintática

- Processando um token.

```
sat :: (s -> Bool) -> Parser s s
sat p = Parser (\ ts -> case ts of
                          [] -> []
                          (t' : ts') ->
                            if p t' then [(t', ts')]
                                    else [])
```

Análise sintática

- Processando uma sequência de tokens.

```
token :: Eq s => [s] -> Parser s [s]
token s = Parser (\ ts -> if (take (length s) ts) == s
                          then [(s, drop (length s) ts)]
                          else [])
```

Análise sintática

- Construção de resultados (instância de Functor)

```
instance Functor (Parser s) where
  fmap f (Parser p) = Parser g
    where g ts = [(f x, ts') | (x, ts') <- p ts]
```

Análise sintática

- Processando um dígito

```
digit :: Parser Char Char
digit = sat isDigit
```

Análise sintática

- Concatenação de parser (instance de Applicative)

```
instance Applicative (Parser s) where
  pure x = Parser (\ ts -> [(x,ts)])
  (Parser pf) <*> (Parser px)
    = Parser (\ ts -> [(f x, ts') | (f, ts1) <- pf ts
                                  , (x, ts') <- px ts1])
```

Análise sintática

- Alternativas de parsers (instance de Alternative)

```
instance Alternative (Parser s) where
  empty = Parser (\ _ -> [])
  (Parser p1) <|> (Parser p2) = Parser f
    where f ts = p1 ts ++ p2 ts
```

Análise sintática

- Repetindo um parser

```
many :: Parser s a -> Parser s [a]
many p = (:) <$> p <*> many p <|> pure []
```

Análise sintática

- Parser para números naturais

```
natural :: Parser Char Int
natural
  = foldl f 0 <$> many digit
    where
      f a b = a * 10 + b
```

Análise sintática

- Executando um parser opcionalmente.

```
option :: Parser s a -> a -> Parser s a
option p v = p <|> pure v
```

Análise sintática

- Parser para números inteiros

```
integer :: Parser Char Int
integer = option (const negate <$> token "-") id <*> natural
```

Análise sintática

- Lidando com separadores.
  - Separadores sem semântica

```
endBy :: Parser s a -> Parser s b -> Parser s [a]
p `endBy` e
  = many (f <$> p <*> e)
    where
      f x _ = x
```

Análise sintática

- Lidando com separadores
  - Separadores com semântica (operadores)

```
chainl :: Parser s (a -> a -> a) -> Parser s a -> Parser s a
chainl op p
  = applyAll <$> p <*> many (flip <$> op <*> p)
    where
      applyAll x [] = x
      applyAll x (f : fs) = applyAll (f x) fs
```

Análise sintática

- De posse de todas essas funções, podemos construir o analisador sintático para a gramática.

$$\begin{array}{lcl} E & \rightarrow & T + E \,|\, T \\ T & \rightarrow & F * T \,|\, F \\ F & \rightarrow & n \,|\, (E) \end{array}$$

Análise sintática

- Antes de construir um parser, precisamos definir o resultado
  - Árvore de sintaxe abstrata.
- Valores

```
data Value
  = VInt Int
```

Análise sintática

- Programas completos: expressões envolvendo adição e multiplicação.

```
data L0
  = LVal Value
  | LAdd L0 L0
  | LMul L0 L0
```

Análise sintática

- Parser de valores

```
valueParser :: Parser Value
valueParser = f <$> sat (\ t -> case lexeme t of
                                  TNumber _ -> True
                                  _ -> False)
      where
        f (Token (TNumber n) _) = n
```

Análise sintática

- Parsing de parêntesis.

```
parens :: Parser a -> Parser a
parens p
  = f <$> lparen <*> p <*> rparen
    where
      f _ x _ = x

lparen :: Parser Token
lparen = sat (\ t -> lexeme t == TLParen)

rparen :: Parser Token
rparen = sat (\ t -> lexeme t == TRParen)
```

Análise sintática

- Parser de fatores

```
factorParser :: Parser L0
factorParser
  = (LVal <$> valueParser) <|> parens expParser
```

Análise sintática

- Parser de termos

```
termParser :: Parser L0
termParser
  = chainl pmult factorParser
    where
      pmult = (const LMul) <$> sat (\ t -> lexeme t == TMul)
```

Análise sintática

- Parser de expressões

```
expParser :: Parser L0
expParser
  = chainl padd termParser
    where
      padd = (const LAdd) <$> sat (\ t -> lexeme t == TPlus)
```

Análise sintática

- Função top-level do analisador sintático

```
l0Parser :: [Token] -> Either String L0
l0Parser tks
  = case runParser expParser tks of
      ((t, []) : _) -> Right t
      _             -> Left "Parse error on program."
```

Intepretador

Interpretador

- De posse de um analisador sintático, podemos agora:
  - Executar o código (interpretador)
  - Gerar código (compilador)

Interpretador

- Intepretador:

```
eval :: L0 -> Either String Value
eval (LVal v) = Right v
eval (LAdd l1 l2)
  = do
       v1 <- eval l1
       v2 <- eval l2
       v1 .+. v2
eval (LMul l1 l2)
  = do
       v1 <- eval l1
       v2 <- eval l2
       v1 .*. v2
```

Interpretador

- Operações sobre valores.

```
(.+.) :: Value -> Value -> Either String Value
(VInt n1) .+. (VInt n2) = Right (VInt (n1 + n2))
e1 .+. e2 = Left $ unwords ["Type error on:", pretty e1, "+", pretty e2]
```

Geração de código

Geração de código

- Vamos agora considerar o problema de gerar código.
  - Para uma máquina virtual
  - Executável, gerando código fonte C, e usar o gcc para produzir o executável.

Geração de código

- Gerar o código C correspondente consiste em:
  - Produzir o código com uma função main.
  - Corpo da função possui uma variável que recebe o resultado da expressão.
  - Imprime o valor da variável usando printf.

Geração de código

- Exemplo:
  - Considere a expressão: (1 + 2) * 3

```
#include <stdio.h>
// code generated for expressions
int main () {
   int val = (1 + 2) * 3;
   printf("%d", val);
   putchar('\n');
   return 0;
}
```

Geração de código

- Como produzir esse código C?
  - Vamos criar funções para produzir a expressão.
  - Usar um "template" do corpo do código C.

Geração de código

- Como produzir o texto da expressão a partir da AST?
  - Essa é a operação inversa da análise sintática
  - Normalmente conhecida como pretty-print

Geração de código

- Para isso, vamos utilizar uma biblioteca Haskell para facilitar essa tarefa.
- Para construir o pretty-print, vamos seguir a estrutura da gramática.
  - Vantagem: evita parêntesis desnecessários.

Geração de código

- Gramática

$$\begin{array}{lcl} E & \rightarrow & T + E \,|\, T \\ T & \rightarrow & F * T \,|\, F \\ F & \rightarrow & n \,|\, (E) \end{array}$$

Geração de código

- Primeiro nível do pretty-print

```
pprAdd :: L0 -> Doc
pprAdd (LAdd e1 e2)
  = hsep [pprAdd e1, text "+", pprAdd e2]
pprAdd other = pprMul other
```

Geração de código

- Segundo nível do pretty-print

```
pprMul :: L0 -> Doc
pprMul (LMul e1 e2)
  = hsep [pprMul e1, text "*", pprMul e2]
pprMul other = pprFact other
```

Geração de código

- Último nível do pretty-print

```
pprFact :: L0 -> Doc
pprFact (LVal v) = ppr v
pprFact other = parens (ppr other)
```

Geração de código

- Gerando o corpo do código C.

```
cL0Codegen :: L0 -> String
cL0Codegen e
  = unlines $ [ "#include <stdio.h>"
              , "// code generated for expressions"
              , "int main () {" ] ++
              (map (nest 3) (generateBody e)) ++
              [ nest 3 $ "putchar('\\n');"
              , nest 3 "return 0;"
              , "}"
              ]
    where
      nest n v = replicate n ' ' ++ v
```

Máquina virtual

Máquina virtual

- Agora vamos considerar a geração de código para uma máquina virtual simples, chamada de V0.

Máquina virtual

- Instruções:
  - Push(n): empilha um valor.
  - Add: desempilha dois valores e empilha a sua soma.
  - Mul: desempilha dois valores e empilha o seu produto.
  - Print: desempilha um valor e o imprime no console.
  - Halt: termina a execução.

Máquina virtual

- Execução de uma instrução, modifica a pilha da máquina.

Máquina virtual

- Executando uma instrução

```
step :: Instr -> Stack -> IO (Either String Stack)
step (Push v) stk = pure (Right (v : stk))
step Add (v1 : v2 : stk)
  = case v1 .+. v2 of
      Left err -> pure (Left err)
      Right v3 -> pure (Right (v3 : stk))
step Print (v : stk)
  = do
      print (pretty v)
      pure (Right stk)
```

Máquina virtual

- Executando uma lista de instruções.

```
interp :: Code -> Stack -> IO ()
interp [] _ = pure ()
interp (c : cs) stk
  = do
      r <- step c stk
      case r of
        Right stk' -> interp cs stk'
        Left err -> putStrLn err
```

Máquina virtual

- Compilando uma expressão

```
codegen' :: L0 -> Code
codegen' (LVal v) = [Push v]
codegen' (LAdd l0 l1)
  = codegen' l0 ++ codegen' l1 ++ [Add]
codegen' (LMul l0 l1)
  = codegen' l0 ++ codegen' l1 ++ [Mul]
```

Máquina virtual

- Compilando uma expressão

```
v0Codegen :: L0 -> Code
v0Codegen e = codegen' e ++ [Print, Halt]
```

Conclusão

Conclusão

- Nesta aula, apresentamos uma implementação para expressões de:
  - Intepretador.
  - Compilador, usando o GCC
  - Compilador para uma máquina virtual.

Tarefa

Tarefa

- Primeira tarefa: obter o ambiente de execução e realizar testes com o código de exemplo.