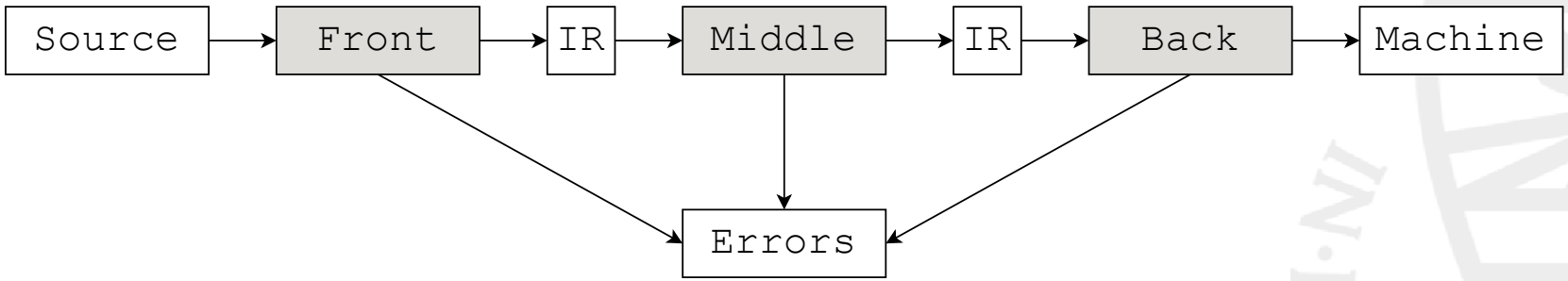
Compiler Construction: Lexical analyses

Quinten Cabo, Marijn van Wezel

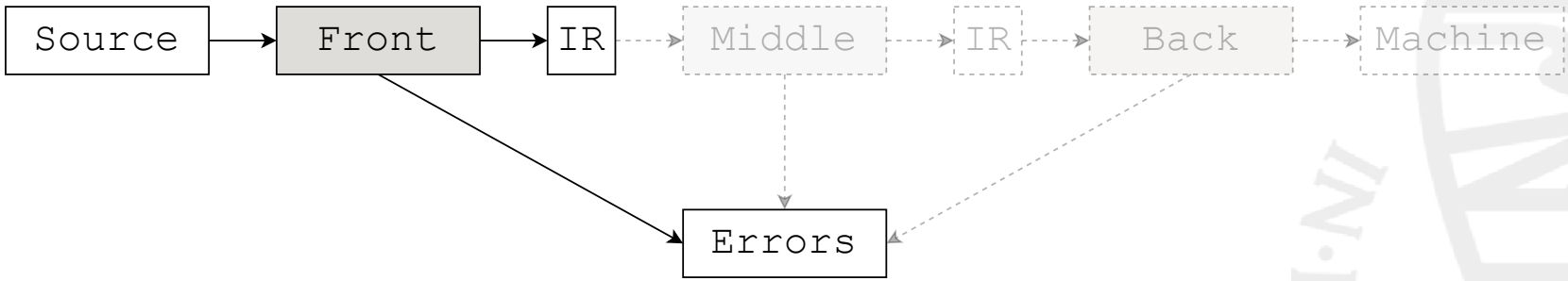
Radboud University Nijmegen

27 February 2024

What was a compiler again?



What was a compiler again?



The goal

- **Given some arbitrary string of characters,**
- analyse this string,
- produce an abstract syntax tree,
- or give an error.

String of characters: Examples

```
sum(l:[Int]): Int {
  if (isEmpty(l.tl)) {
    return l.hd;
  }

  return l.hd + sum(l.tl);
}
```

String of characters: Examples

```
product(list:[Int]): Int {
  if (isEmpty(list)) {
    return 1;
  }

  return list.hd * product(list.tl);
}
```

String of characters: Examples

```
product(list:[Int]: Int {
  if (isEmpty(list)) {
    return 1;
  }

  return list.hd * product(list.tl);
}
```

The goal

- Given some arbitrary string of characters,
- **analyse this string,**
- produce an abstract syntax tree,
- or give an error.

Analysis: Haskell

We chose *Haskell* to implement our parser (and the rest of our compiler):

- Algebraic data types;
- High level of abstraction;
- Good tooling and library support;
- **Functional parser combinators**.

Analysis: Haskell

We chose *Haskell* to implement our parser (and the rest of our compiler):

- Algebraic data types;
- High level of abstraction;
- Good tooling and library support;
- **Functional parser combinators**.

Analysis: Haskell

We chose *Haskell* to implement our parser (and the rest of our compiler):

- Algebraic data types;
- High level of abstraction;
- Good tooling and library support;
- **Functional parser combinators**.

Analysis: Haskell

We chose *Haskell* to implement our parser (and the rest of our compiler):

- Algebraic data types;
- High level of abstraction;
- Good tooling and library support;
- **Functional parser combinators**.

Analysis: Functional parser combinators

Parser combinators work through *function composition*.

Analysis: Megaparsec

The Andromeda Galaxy is roughly one **megaparsec** away (Torben Hansen, CC-BY-2.0).

Analysis: Megaparsec

- Informal successor of Parsec;
- Does much of the heavy lifting;
- Large collection of combinators;
- Decent error messages out-of-the-box;
- `makeExprParser`

Analysis: Megaparsec

- Informal successor of Parsec;
- Does much of the heavy lifting;
- Large collection of combinators;
- Decent error messages out-of-the-box;
- `makeExprParser`

Analysis: Megaparsec

- Informal successor of Parsec;
- Does much of the heavy lifting;
- Large collection of combinators;
- Decent error messages out-of-the-box;
- `makeExprParser`

Analysis: Megaparsec

- Informal successor of Parsec;
- Does much of the heavy lifting;
- Large collection of combinators;
- Decent error messages out-of-the-box;
- `makeExprParser`

Analysis: Megaparsec

- Informal successor of Parsec;
- Does much of the heavy lifting;
- Large collection of combinators;
- Decent error messages out-of-the-box;
- `makeExprParser`

Analysis: Tokenless parsing

Analysis: Generating expression parsers

Analysis: Testing the parser

hspec example:

```
describe "pTrue" $ do
  it "parses the string 'true' as TrueLit" $ do
      parse pTrue "test.spl" "false" `shouldParse` TrueLit

  it "doesn't parse anything else" $ do
      parse pTrue "test.spl" `shouldFailOn` "false"
      parse pTrue "test.spl" `shouldFailOn` "fa"
      parse pTrue "test.spl" `shouldFailOn` "tru"
```

Analysis: Testing the parser output

```
pTrue
  parses the string 'true' as TrueLit [x]

Failures:

  app/Test/Parser/Expr.hs:132:48:
  1) Parser.Parser.Parser.Expr.pTrue parses
     the string 'true' as TrueLit
       expected: TrueLit
       but parsing failed with error:
         test.spl:1:1:
           |
         1 | false
           | ^^^^
         unexpected "fals"
         expecting "true"
```

Analysis: Testing the parser

```
describe "pTrue" $ do
  it "parses the string 'true' as TrueLit" $ do
      parse pTrue "test.spl" "true" `shouldParse` TrueLit
```

Analysis: Testing the parser

```
  pTrue
    parses the string 'true' as TrueLit [v]
    doesn't parse anything else [v]
  pFalse
    parses the string 'true' as TrueLit [v]
    doesn't parse anything else [v]
  pInt
    parses an integer [v]
    parses a signed integer [v]
    discards trailing whitespace [v]

Finished in 0.0091 seconds
88 examples, 0 failures
```

The goal

- Given some arbitrary string of characters,
- analyse this string,
- **produce an abstract syntax tree,**
- or give an error.

Abstract syntax tree

```
type Program = undefined
data Type = undefined
data Decl = undefined
data Stmt = undefined
data Expr = undefined
data UnaryOp = undefined
data BinOp = undefined
data Variable = undefined
data Literal = undefined
% Field
```

Abstract syntax tree

```
data Literal =
      TrueLit                  -- true
    | FalseLit                 -- false
    | IntLit Int               -- 10, -10, +10
    | FloatLit Float           -- 10.0, -10.0, +10.0
    | CharLit Char             -- 'a'
    | TupleLit (Expr, Expr) -- (a, b)
    | EmptyListLit             -- []
    deriving (Eq, Show)
```

Abstract syntax tree

```
data Expr =
      BinOp BinOp Expr Expr       -- a + b
    | UnaryOp UnaryOp Expr        -- + a
    | AssignExpr Variable Expr    -- a = b
    | FunctionCall String [Expr] -- f()
    | VariableExpr Variable       -- a, a.tl, a.hd
    | LiteralExpr Literal         -- 10
    deriving (Eq, Show)
```

Abstract syntax tree

```
data UnaryOp =
    Negate                -- !a
  | FieldAccess Field     -- .hd, .tl
  deriving (Eq, Show)
```

```
data BinOp =
      Mul    -- '*'
    | Div    -- '/'
    | Mod    -- '\%'
    | Add    -- '+'
    | Sub    -- '-'
    | Cons   -- ':'
    | Gt     -- '>'
    | Gte    -- '>='
    | Lt     -- '<'
    | Lte    -- '<='
    | Eq     -- '=='
    | Neq    -- '!='
    | And    -- '\&\&'
    | Or     -- '||'
  deriving (Eq, Show)
```

Abstract syntax tree

```
data Variable =
      Identifier String (Maybe Field) -- a, a.hd, a.tl
    deriving (Eq, Show)

data Field =
      HeadField -- hd
    | TailField -- tl
    deriving (Eq, Show)
```

Abstract syntax tree

```
data Stmt =
  ReturnStmt (Maybe Expr)                -- return a;
| IfStmt Expr [Stmt] (Maybe [Stmt])      -- if (a) {b} else {c}
| WhileStmt Expr [Stmt]                  -- while (a) {b}
| ExprStmt Expr                          -- a;
| VarStmt (Maybe Type) String Expr       -- var hello = 'h':'i':[]
deriving (Eq, Show)
```

Abstract syntax tree

```
data Stmt =
  ReturnStmt (Maybe Expr)
| IfStmt Expr [Stmt] (Maybe [Stmt])
| WhileStmt Expr [Stmt]
| ExprStmt Expr
| VarStmt (Maybe Type) String Expr
deriving (Eq, Show)
```

```
data Type =
  IntType              -- Int
| CharType             -- Char
| BoolType             -- Bool
| VoidType             -- Void
| TupleType Type Type  -- (a, b)
| ListType Type        -- [a]
| TypeVar String       -- a
deriving (Eq, Show)
```

Abstract syntax tree

```
-- a(b : c, d : e) : Bool {
--     f();
-- }
data Decl =
  FunDecl String (Maybe Type) [(String, Maybe Type)] [Stmt]
  deriving (Eq, Show)
```

Abstract syntax tree

```
-- a(b : c, d : e) : Bool {
--     f();
-- }
data Decl =
  FunDecl String (Maybe Type) [(String, Maybe Type)] [Stmt]
  deriving (Eq, Show)


type Program = [Decl]
```

Abstract syntax tree - example

```
fib(n : Int) : Int {
  if (n == 0) {
      return 0;
  }
  if (n == 1) {
      if (true) {
          print('h':'i':[]);
      }
      return 1;
  }
  return fib(n - 1) + fib(n - 2);
}
```

Abstract syntax tree - example

```
[FunDecl "fib" (Just IntType) [("n",Just IntType)]
 [IfStmt (BinOp Eq (VariableExpr (Identifier "n" Nothing))
         (LiteralExpr (IntLit 0)))
         [ReturnStmt (Just (LiteralExpr (IntLit 0)))] Nothin
  IfStmt (BinOp Eq (VariableExpr (Identifier "n" Nothing))
         (LiteralExpr (IntLit 1)))
         [IfStmt (LiteralExpr TrueLit) [
            ExprStmt (FunctionCall "print" [BinOp Cons (Lite
             (BinOp Cons (LiteralExpr (CharLit 'i'))
              (LiteralExpr EmptyListLit))])] Nothing,
          ReturnStmt (Just (LiteralExpr (IntLit 1)))] Nothin
           ReturnStmt (Just (BinOp Add (FunctionCall "fib"
                [BinOp Sub (VariableExpr (Identifier "n" Not
                (LiteralExpr (IntLit 1))])
                (FunctionCall "fib" [BinOp Sub
                (VariableExpr (Identifier "n" Nothing))
                (LiteralExpr (IntLit 2))]))))]]
```

The goal

- Given some arbitrary string of characters,
- analyse this string,
- produce an abstract syntax tree,
- **or give an error.**

Error reporting

Pretty printer

```
    fib(n : Int) : Int {
      if (n == 0) {return 0;}


if    (n == 1) {
   if           (true)          {print('h':'i':[]);}
return 1;}return (fib(n - 1) + fib(n - 2));  }
```

Pretty printer

```
    fib(n : Int) : Int {
      if (n == 0) {return 0;}


if    (n == 1) {
   if            (true)           {print('h':'i':[]);}
return 1;}return (fib(n - 1) + fib(n - 2));  }
```

Pretty printer

```
fib(n : Int) : Int {
  if (n == 0) {
      return 0;
  }
  if (n == 1) {
      if (true) {
          print('h':'i':[]);
      }
      return 1;
  }
  return fib(n - 1) + fib(n - 2);

}
```

Pretty printer - Comments

```
    fib(n : Int) : Int {
      if (n == 0) {return 0;}

/*    /| _________________
0|===|* >_________________>
      \|                          */

if    (n == 1) { //                                  h   i
   if /* always true? */  (true) /* hi */    {print('h':'i':[]);
return 1;}return (fib(n - 1) + fib(n - 2));  }
```

Pretty printer - Comments

```
fib(n : Int) : Int {
  if (n == 0) {
      return 0;
  }
  if (n == 1) {
      if (true) {
          print('h':'i':[]);
      }
      return 1;
  }
  return fib(n - 1) + fib(n - 2);

}
```

Pretty printer - Comments

```
type Program = ([Decl], [Comment])
                      -- start  end
data Comment = Single Int Int Int Int String
             | Multi  Int Int Int Int String
% Make the symbol parser fix it
```

Questions?