



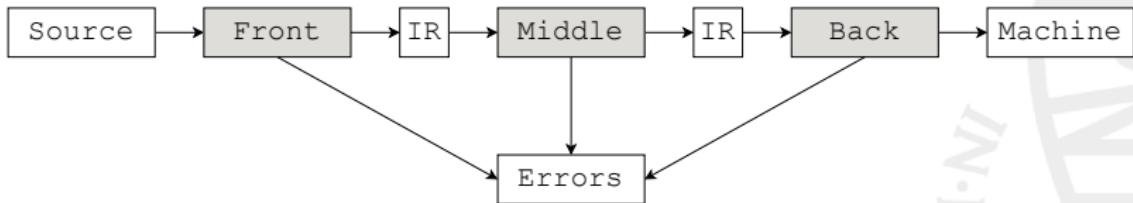
Compiler Construction: Lexical analyses

Quinten Cabo, Marijn van Wezel

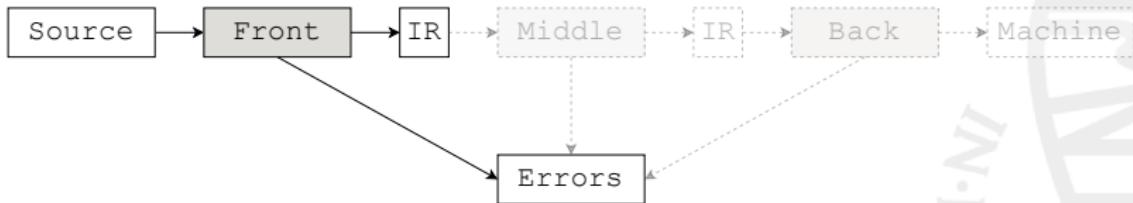
Radboud University Nijmegen

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.



The goal

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



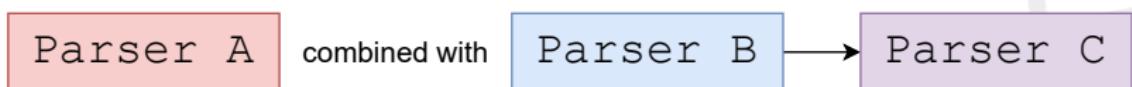
Analysis: Haskell

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: 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: Tokenless parsing

We decided to use a tokenless (lexer-less) parsing approach.

- Works well with parser combinators
- Good support by megaparsec

Analysis: Tokenless parsing

Tokens are parsed using special symbol parsers.

-- ...

```
tTrue :: Parser T.Text  
tTrue = symbol "true"
```

```
tFalse :: Parser T.Text  
tFalse = symbol "false"
```

```
tEmptyList :: Parser T.Text  
tEmptyList = symbol "[]"
```

-- ...

Analysis: Tokenless parsing

Token parsers strip whitespace (spaces, tabs, comments) at the end of the input:

```
whitespace = L.space
space1
(L.skipLineComment("//"))
(L.skipBlockComment /* */)

tColon :: Parser T.Text
tColon = symbol ":"  
  
-- tColon parses ":", but also ": /* Parsers are fun */"
```

Analysis: Tokenless parsing

```
pProgram :: Parser Program
pProgram = L.whitespace *> some pDecl
```

Analysis: Generating expression parsers

```
operatorTable :: [[Operator Parser Expr]]
operatorTable =
  [ [ Postfix (UnaryOp (FieldAccess HeadField)
    <$ try (L.tDot <* L.tHead))
    , Postfix (UnaryOp (FieldAccess TailField)
    <$ try (L.tDot <* L.tTail))
  ]
  , ...
  ]

pTerm :: Parser Expr
pTerm = choice
  [ try $ L.parens pExpr
  , try pFunctionCall
  , try pAssignExpr
  , try pLiteralExpr
  , try pVariableExpr
  ]

pExpr :: Parser Expr
pExpr = makeExprParser pTerm operatorTable
```



Analysis: Testing the parser

```
exprSpec :: Spec
exprSpec = do
    describe "Parser.Expr" $ do
        describe "pExpr" $ do
            it "parses a term" $ do
                parse pExpr "test.spl" "1"
                `shouldParse` LiteralExpr (IntLit 1)
                parse pExpr "test.spl" "a = 'b'"
                `shouldParse` AssignExpr (Identifier "a" Nothing) (LiteralExpr $ CharLit 'b')
                parse pExpr "test.spl" "ident"
                `shouldParse` VariableExpr (Identifier "ident" Nothing)
                parse pExpr "test.spl" "(a + b)"
                `shouldParse` BinOp Add
                    (VariableExpr (Identifier "a" Nothing))
                    (VariableExpr (Identifier "b" Nothing))
```

Analysis: Testing the parser

Parser.Expr

pExpr

parses a term [v]

parses a simple unary expression [v]

parses a simple binary expression [v]

parses an expression [v]

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

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"
```

The goal

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



The goal

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



Abstract syntax tree: design goals

- We started with the design of our AST, and based our parser on that;
- Tried to make the AST only be able to represent syntactically valid programs;
- Tried to make the AST as simple as possible;
- `data` type is Haskell.

Abstract syntax tree: The Skeleton

```
type Program = [Decl]
data Decl = ...
data Type = ...
data Stmt = ...
data Expr = ...
data UnaryOp = ...
data BinOp = ...
data Variable = ...
data Field = ...
data Literal = ...
```



Abstract syntax tree: example

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



The goal

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



The goal

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



Error reporting

Megaparsec generates decent error messages out of the box:

```
examples/product_parse_error.spl:1:19:  
1 | product(list:[Int]: Int {  
|  
unexpected ':'  
expecting ')' or ','
```

But also has support for custom error messages, and even error recovery. We have not yet implemented this.

Side quest: A pretty printer



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) { //  
        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
```

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?

