



LEXING

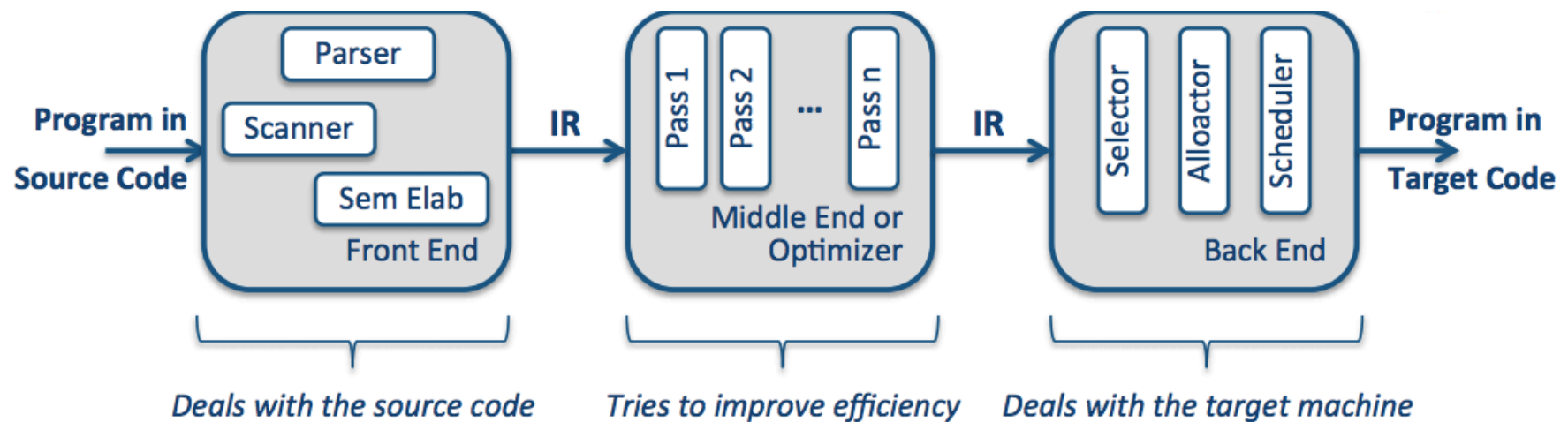
cs4430/7430 Spring 2019

Bill Harrison

Announcements

- "CS4430 Code Repository" is a thing:
 - <https://bitbucket.org/william-lawrence-harrison/cs4430>
- "Homework 0": install the Haskell Platform, if you haven't already.

Earliest Phase: Scanning a.k.a. Lexing



The "Three Address Code" Language

- Here's a program in the ThreeAddr language
 - ...the intermediate representation used in the Imp compiler
- This program is in **concrete syntax**
 - i.e., the syntax that we (i.e., us humans) use to write a program

```
mov R0 #99;  
mov Rx R0;  
0: mov R1 #0;  
   sub R2 Rx R1;  
   brnz R2 #2;  
   mov R2 #0;  
   jmp #3;  
2: mov R2 #1;  
3: brz R2 #1;  
   mov R3 #1;  
   sub R4 Rx R3;  
   mov Rx R4;  
   jmp #0;  
1:
```

"Three Address Code Language" (also)

- This is also the Three Address Code language
 - ...as **abstract syntax**
- Abstract syntax is the representation of the language used by the compiler

```
data ThreeAddrProg
    = ThreeAddrProg [ThreeAddr]
```

```
data ThreeAddr
    = Mov Register Arg
    | Load Register Register
    | Store Register Register
    ...
    | Call Arg
    | Ret
    | Exit
```

```
data Register
    = Reg String | SP | FP | BP
```

```
data Arg = Immediate Register |
          Literal Word
```

Front End Types

```
front3addr :: String -> Maybe [ThreeAddr]
front3addr = lexer <> parse3addr
```

```
lexer :: String -> Maybe [Token]
```

```
parse3addr :: [Token] -> Maybe [ThreeAddr]
```

```
(<>) :: Monad m => (a -> m b) ->
                  (b -> m c) ->
                  a -> m c
```

```
f <> g = \ a -> f a >>= g
```

Running the front end

With `Show` instances

```
λ> front3addr foobar  
Just [mov R0 #99,...,jmp #0,1:]
```

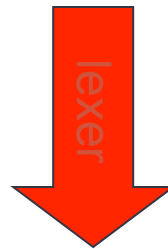
Without `Show` instances

```
λ> front3addr foobar  
Just [Mov (Reg "0") (Literal 99),  
      ...  
      Jmp (Literal 0),Label 1]
```

Front End: Lexical Analysis

ascii form

c	l	a	s	s		p	u	b	l	i	c		F	o	o		{		i	n	t	...
---	---	---	---	---	--	---	---	---	---	---	---	--	---	---	---	--	---	--	---	---	---	-----



"tokens"

class	public	name("Foo")	left-brack	type-int	...
-------	--------	-------------	------------	----------	-----

What are the tokens for ThreeAddr?

Tokens for ThreeAddr

```
    mov R0 #99;
    mov Rx R0;
0:   mov R1 #0;
    sub R2 Rx R1;
    brnz R2 #2;
    mov R2 #0;
    jmp #3;
2:   mov R2 #1;
3:   brz R2 #1;
    mov R3 #1;
    sub R4 Rx R3;
    mov Rx R4;
    jmp #0;
1:
```

```
data Token = MOV | LOAD | STORE
           | ADD | SUB | DIV | MUL
           | NEGATE | EQUAL | NOT
           | GTHAN | JMP | BRZ | BRNZ
           | BRGT | BRGE | READ
           | WRITE | CALL | RET
           | EXIT | REG String
           | LIT Int | FPtok | SPtok
           | BPtok | SEMICOL | COLON
           | ENDOFINPUT
```

```
λ> lexer foobar
      Just [MOV,REG "0",LIT 99,SEMICOL,
            MOV,REG "x",REG "0",SEMICOL,
            LIT 0,...,ENDOFINPUT]
```

The Lexer

```
lexer :: String -> Maybe [Token]
lexer [] = return [ENDOFINPUT]
lexer ('/':'/':cs) = consumeLine cs
lexer (c:cs)
  | isSpace c = lexer cs
  | isAlpha c = lexAlpha (c:cs)
  | isDigit c = lexNum (c:cs)
  | c==';'    = do
      rest <- lexer cs
      return $ SEMICOL : rest
  | c==':'    = do
      rest <- lexer cs
      return $ COLON : rest
  | c=='#'    = lexNum cs
  | otherwise = Nothing
```

Notation Alert!

`f $ g x` is
`f (g x)`

do?
return?

what input
might
generate
Nothing?

Errors

- Errors are an important aspect of computation.
- They are typically a pervasive feature of a language, because they affect the way every expression is evaluated. For example, consider the expression:

$a + b$

- If a or b raise errors then we need to deal with this possibility.
- Lexical errors include unrecognized symbols

Errors

- Because errors are so pervasive they are a notorious problem in programming and programming languages.
- When coding in C the convention is to check the return codes of all system calls.
- However this is often not done.
 - Java's exception handling mechanism provides a more robust way to deal with errors.
- Errors are a kind of "side effect"
 - Therefore, they are encoded as a "Monad" in Haskell

Maybe

- The Maybe datatype provides a useful mechanism to deal with errors:

```
data Maybe a = Nothing | Just a
```



Error!



Good result!

Monads in Haskell

- Monads are a structure composed of two basic operations (**bind** and **return**), which capture a common pattern that occurs in many types.
- In Haskell Monads are implemented using type classes:

```
class Monad m where
    (>>=)    :: m a -> (a -> m b) -> m b
    return  :: a -> m a
```

Maybe as a Monad

Because `Maybe` can implement `return` and `bind` it can be made an instance of `Monad`

```
instance Monad Checked where
  return v = Just v
  x >>= f  = case x of
               Nothing -> Nothing
               Just v   -> f v
```

Do-notation

- However, because monads are so pervasive, Haskell supports a special notation for monads (called the **do-notation**).
- Using do-notation, write lexer as follows:

```
| c == ';' = do
    rest <- lexer cs
    return $ SEMICOL : rest
```


Do-notation

- In Haskell, code using the do-notation, such as:

```
do  pattern <- exp  
    morelines
```

Is converted to code using this transformation:

```
exp >>= (\pattern -> do morelines)
```

Monad Laws

- It is not enough to implement `bind` and `return`. A proper monad is also required to satisfy some laws:

$$\text{return } a \gg= k == k \ a$$
$$m \gg= \text{return} == m$$
$$\begin{aligned} m \gg= (\backslash x \rightarrow k \ x \gg= h) \\ == (m \gg= k) \gg= h \end{aligned}$$

Maybe

- However, sometimes we would like to track some more information about what went wrong.
- For example, perhaps we would like to report an error message.
- The Maybe datatype is limiting in this case, because Nothing does not track any information.
- How to improve the Maybe datatype to allows us to track more information?

Representing Errors

- We can create a datatype `Checked`, provides a constructor `Error` to be used instead of `Nothing`

```
data Checked a = Good a | Error String
```



A good value!



Error with an
error message!

Checked as a Monad

Because `Checked` can implement `return` and `bind` it can be made an instance of `Monad`

```
instance Monad Checked where
  return v = Good v
  x >>= f  =
    case x of
      Error msg -> Error msg
      Good v    -> f v
```