CS 4450: Principles of Programming Languages

10/24/2018

Languages with Effects

Today

- Backus-Naur Form (BNF)
- We'll consider how to write an interpreter for a small arithmetic language

BNF derivations

Given a BNF grammar, we use it to show that certain strings are in the language (and others are not!)

1. <LON> ::= ()
2. <LON> ::= (<number> **.** <LON>)

Ex: show (12 . ()) ∈ <LON>

<LON> => (<number> . <LON>) { from 2 }
=> (<number> . ()) { from 1 }
=> (12 . ()) { 12 ∈ <number> }

BNF derivations

- **General rule:** string s∈<L> if and only if there is a sequence of production steps: <L> => … => s
  - Note that there may be more than one such sequence
  - gives a rigorous definition for the syntax of a (programming) language
- This is the **parsing** problem

BNF for programming language syntax: the λ-calculus

<expr> ::= <ident>
<expr> ::= (`lambda` (<ident>) <expr>)
<expr> ::= (<expr> <expr>)

where <ident> is any symbol other than "`lambda`":
`+ - abc x y if cdr null?` …

BNF for programming language syntax

<expr> ::= <ident>
<expr> ::= (`lambda` (<ident>) <expr>)
<expr> ::= (<expr> <expr>)

Q: is the following Scheme program in <expr>?

```
(lambda (l)
    (if (null? l)
        0
        (+ 1 (length (cdr l)))))
```

```
(lambda (l)
    (if (null? l)
        0
        (+ 1 (length (cdr l)))))
```

1. <expr> ::= <ident>
2. <expr> ::= (lambda (<ident>) <expr>)
3. <expr> ::= (<expr> <expr>)

<expr> → (lambda (<ident>) <expr>)
→ (lambda (l) <expr>)
→ (lambda (l) (<expr> <expr>))
→ (lambda (l) (<ident> <expr>))
→ (lambda (l) (if <expr>))

```
(lambda (l)
   (if (null? l)
       0
       (+ 1 (length (cdr l)))))
```

1. <expr> ::= <ident>
2. <expr> ::= (`lambda` (<ident>) <expr>)
3. <expr> ::= (<expr> <expr>)

<expr> → (`lambda` (<ident>) <expr>)
→ (`lambda` (`l`) <expr>)
→ (`lambda` (`l`) (<expr> <expr>))
→ (`lambda` (`l`) (<ident> <expr>))
→ (`lambda` (`l`) (`if` <expr>))

`(null? l) 0 (+ 1 (length (cdr l)))` ?

Syntax for Basic Scheme

<expr> ::= <ident>
<expr> ::= (`lambda` (<ident>*) <expr>)
<expr> ::= (<expr>*)

How might we represent this as a Haskell datatype?

Syntax for Basic Scheme

<expr> ::= <ident>
<expr> ::= (`lambda` (<ident>*) <expr>)
<expr> ::= (<expr>*)

```
data Expr = Ident String
          | Lambda [String] Expr
          | Funcall [Expr]
```

Today: Processing Languages with Effects

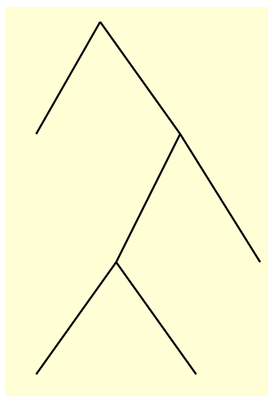
- Representing languages: Abstract Syntax
  - …and how to represent AS in Haskell
    - Defining new types with `data` declarations
  - Abstract Syntax Trees
  - Introduction to Backus-Naur Form (BNF)
    - Wikipedia entry may be helpful
      - http://en.wikipedia.org/wiki/Backus-Naur_form
- Defining Languages with Interpreters
  - Simple example: arithmetic language
- Key consideration: languages with "effects"
  - Ex: how do you interpret "1 / 0"?

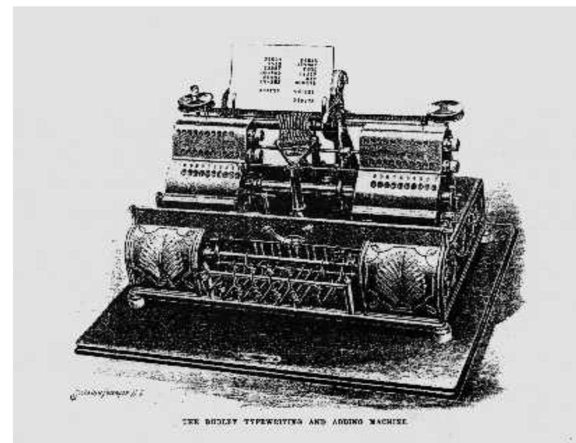
We considered syntax last time

An interpreter is a function

Representation
of a program

Each expression/statement
defined in terms of Haskell





`interp :: AbstractSyntax` ⟶ `“Computation of Values”`

The Language We'll Interpret Today

```
data Op    = Plus | Minus | Times | Div
data Exp   = Const Int | Aexp Op Exp Exp
```

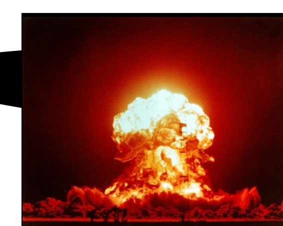
N.b., simpler than Scheme;
all its operators
are <u>binary</u>, and only Int values

```
ex1 = Aexp Plus (Const 1) (Const 2) -- (+ 1 2)

ex2 = Aexp Div (Const 1) (Const 0)  -- (/ 1 0)
```

Interpreter, v1.0

```
interp :: Exp -> Int
interp (Const v)            = v
interp (Aexp Plus e1 e2)    = interp e1 + interp e2
interp (Aexp Minus e1 e2) = interp e1 - interp e2
interp (Ae                                interp e2
interp (Ae                               iv` interp e2
```

```
Arith> interp ex1
3
Arith> interp ex2

Program error: divide by zero
```



An error is a “side effect”

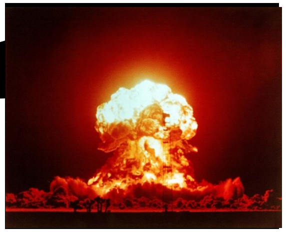
The value of an Exp is an Int, but
the error “happens on the side” - i.e., it’s not a proper value

```
Arith> interp ex2

Program error: divide by zero
```

Question: how do we handle this error side effect?
Answer: Add a value for errors

Haskell has a tool for just this sort of thing:

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



Handling an error

Haskell has a tool for just this sort of thing:

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

- If (interp e) doesn't have any errors, then return (Just v) where v is the value of e
- Otherwise, signify that something is wrong within (interp e) by returning Nothing

This is what we want

```
Arith> interp ex1
Just 3
Arith> interp ex2
Nothing
```

Interpreter, v2.0

```
interp2 :: Exp -> Maybe Int
interp2 (Const v)            = Just v
interp2 (Aexp Plus e1 e2)
 = case (interp2 e1,interp2 e2) of
        (Just v1,Just v2) -> Just (v1 + v2)
        (_,_)             -> Nothing
interp2 (Aexp Div e1 e2)
 = case (interp2 e1,interp2 e2) of
        (Just v1,Just 0)  -> Nothing
        (Just v1,Just v2) -> Just (v1 `div` v2)
        (_,_)             -> Nothing
```

notice how the fact that errors may happen is now reflected in the type signature

This works, but…

- The definition is a little bit “hairy”
  - i.e., there’s a lot of exposed “plumbing”
- We want to have our cake and eat it, too
  - accurate error handling
  - plus easy-to-read code
- Haskell has tools for this
  - called “do notation”
  - we’ll see another version later

do Notation

```
interp3 (Aexp Plus e1 e2) = do v1 <- interp3 e1
                               v2 <- interp3 e2
                               Just (v1+v2)
```

- evaluate (interp3 e1) first; if it is:
  - (Just v1), then strip off the Just,
  - Nothing, then the whole do expression is Nothing
- evaluate (interp3 e2) next; if it is:
  - (Just v2), then strip off the Just,
  - Nothing, then the whole do expression is Nothing
- If you’ve got both v1 and v2, Just (v1+v2)

Interpreter, v3.0

```
interp3 :: Exp -> Maybe Int
interp3 (Const v)           = Just v
interp3 (Aexp Plus e1 e2) = do v1 <- interp3 e1
                                v2 <- interp3 e2
                                Just (v1+v2)
interp3 (Aexp Div e1 e2)  = do v1 <- interp3 e1
                                v2 <- interp3 e2
                                if v2==0
                                  then
                                    Nothing
                                  else
                                    Just (v1+v2)
```

Alternative formulation of “do”

```
do v <- x
   e
```

can also be written

“bind”

```
x >>= \ v -> e
```

Ex:

```
do v1 <- interp3 e1
   v2 <- interp3 e2
   Just (v1+v2)
```

becomes

```
interp3 e1 >>= \ v1 ->
interp3 e2 >>= \ v2 ->
   Just (v1+v2)
```

Another equivalence: can write “**`return`**” for “**`Just`**” as:

```
Just (v1+v2) == return (v1+v2)
```

Interpreter, v4.0

```
interp4 :: Exp -> Maybe Int
interp4 (Const v)            = Just v
interp4 (Aexp Plus e1 e2)    = interp4 e1 >>= \ v1 ->
                               interp4 e2 >>= \ v2 ->
                               Just (v1+v2)
interp4 (Aexp Minus e1 e2)   = interp4 e1 >>= \ v1 ->
                               interp4 e2 >>= \ v2 ->
                               Just (v1-v2)
interp4 (Aexp Times e1 e2)   = interp4 e1 >>= \ v1 ->
                               interp4 e2 >>= \ v2 ->
                               Just (v1*v2)
interp4 (Aexp Div e1 e2)     = interp4 e1 >>= \ v1 ->
                               interp4 e2 >>= \ v2 ->
                                  if v2==0
                                     then Nothing
                                     else Just (v1 `div` v2)
```

Throwing an error

```
interp4 (Aexp Div e1 e2) = interp4 e1 >>= \ v1 ->
                           interp4 e2 >>= \ v2 ->
                              if v2==0
                                then Nothing
                                else Just (v1 `div` v2)
```

Pattern

```
if condition
   then throw_error
   else good_value
```

generalizes to:

```
throw :: ?
throw condition goodval = if condition
                          then Nothing
                          else Just goodval
```

Throwing an error

```
interp4 (Aexp Div e1 e2) = interp4 e1 >>= \ v1 ->
                           interp4 e2 >>= \ v2 ->
                           throw (v2==0) (v1 `div` v2)
```

Pattern

```
if condition
   then throw_error
   else good_value
```

generalizes to:

```
throw :: ?
throw condition goodval = if condition
                            then Nothing
                            else Just goodval
```

Throwing an error

```
interp4 (Aexp Div e1 e2) = interp4 e1 >>= \ v1 ->
                           interp4 e2 >>= \ v2 ->
                           throw (v2==0) (v1 `div` v2)
```

Pattern

if *condition*
  then *throw_error*
  else *good_value*

generalizes to:

```
throw :: Bool -> a -> Maybe a
throw condition goodval = if condition
                            then Nothing
                            else Just goodval
```

Interpreter, v5.0

```
interp5 :: Exp -> Maybe Int
interp5 (Const v)            = return v
interp5 (Aexp Plus e1 e2)    = interp5 e1 >>= \ v1 ->
                               interp5 e2 >>= \ v2 ->
                               return (v1+v2)
interp5 (Aexp Minus e1 e2)   = interp5 e1 >>= \ v1 ->
                               interp5 e2 >>= \ v2 ->
                               return (v1-v2)
interp5 (Aexp Times e1 e2)   = interp5 e1 >>= \ v1 ->
                               interp5 e2 >>= \ v2 ->
                               return (v1*v2)
interp5 (Aexp Div e1 e2)     = interp5 e1 >>= \ v1 ->
                               interp5 e2 >>= \ v2 ->
                               throw (v2==0) (v1 `div` v2)
```

Observe: Just and Nothing don’t occur in the above text

Bind & return are overloaded

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

instance Monad Maybe where
  Nothing  >>= f  = Nothing
  (Just v) >>= f  = f v
  return v        = Just v
```

Many important instances of **Monad** class:
- IO monad for input/output
- Lists are a monad
- Monads are used to model “side effects”

Conclusion

- All of this code is available from my website
  - Arith1.hs,…, Arith5.hs
  - Take a look at it to get familiar with Maybe as a monad.
- Side Effects
  - Errors are "to the side" of the Int values produced by interp functions
  - Other kinds of side effects?