Parsing, Concluded

CS 132: Compiler Design

February 2, 2011

Dangling Else

```
S -> if E then S else S
   | if E then S
   | ...
```

How to parse

```
if a then if b then s1 else s2
```

Dangling Else

```
S -> if E then S else S
   | if E then S
   | ...
```

How to parse

```
if a then if b then s1 else s2
```

No ambiguous grammar is LR(k), so you'd get a shift-reduce conflict

```
S -> if E then S . else S     <any>
S -> if E then S .            else
```

Solutions

Hack the grammar

```
S -> M
S -> U

M -> if E then M else M
M -> ...

U -> if E then S
U -> if E then M else U
```

Solutions

Hack the grammar

```
S -> M
S -> U

M -> if E then M else M
M -> ...

U -> if E then S
U -> if E then M else U
```

Hack the language

```
S -> if E then S else S endif
   | if E then S endif
   | ...
```

Solutions

Hack the grammar

```
S -> M
S -> U

M -> if E then M else M
M -> ...

U -> if E then S
U -> if E then M else U
```

Hack the language

```
S -> if E then S else S endif
   | if E then S endif
   | ...
```

Hack the parser:
Whenever you reach this "conflicting" state, just go ahead and shift.

Ambiguous Expression Grammars

```
E -> E + E | E - E
   | E * E | E / E
   | ( E ) | n
```

Ambiguous Expression Grammars

```
E -> E + E | E - E
   | E * E | E / E
   | ( E ) | n
```

All sorts of shift-reduce conflicts:

```
E -> E . + E    <any>
E -> E * E .    +
```

```
E -> E . * E    <any>
E -> E + E .    *
```

```
E -> E . + E    <any>
E -> E + E .    +
```

Solutions

Hack the grammar

```
E -> E + T | E - T | T      E -> T (+ T | - T)*
T -> T * F | T / F | F      T -> F (* F | / F)*
F -> ( E ) | n              F -> ( E ) | n
```

Solutions

Hack the grammar

```
E -> E + T | E - T | T     E -> T (+ T | - T)*
T -> T * F | T / F | F     T -> F (* F | / F)*
F -> ( E ) | n             F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Solutions

Hack the grammar

```
E -> E + T | E - T | T      E -> T (+ T | - T)*
T -> T * F | T / F | F      T -> F (* F | / F)*
F -> ( E ) | n              F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Hack the parser to shift or reduce as desired.

```
E -> E . + E     <any>
E -> E * E .     +
```

```
E -> E . * E     <any>
E -> E + E .     *
```

```
E -> E . + E     <any>
E -> E + E .     +
```

Solutions

Hack the grammar

```
E -> E + T | E - T | T      E -> T (+ T | - T)*
T -> T * F | T / F | F      T -> F (* F | / F)*
F -> ( E ) | n              F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Hack the parser to shift or reduce as desired. YACC support:

```
%left PLUS MINUS
%left STAR SLASH
```

Solutions

Hack the grammar

```
E -> E + T | E - T | T      E -> T (+ T | - T)*
T -> T * F | T / F | F      T -> F (* F | / F)*
F -> ( E ) | n              F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Hack the parser to shift or reduce as desired. YACC support:

```
%left PLUS MINUS
%left STAR SLASH
```

✓ Terminals listed in increasing precedence

Solutions

Hack the grammar

```
E -> E + T | E - T | T     E -> T (+ T | - T)*
T -> T * F | T / F | F     T -> F (* F | / F)*
F -> ( E ) | n             F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Hack the parser to shift or reduce as desired. YACC support:

```
%left PLUS MINUS
%left STAR SLASH
```

- ✓ Terminals listed in increasing precedence
- ✓ Precedence of a rule is that of its last terminal

Solutions

Hack the grammar

```
E -> E + T | E - T | T      E -> T (+ T | - T)*
T -> T * F | T / F | F      T -> F (* F | / F)*
F -> ( E ) | n              F -> ( E ) | n
```

Hack the language

```
E -> + E E | * E E | ...
```

Hack the parser to shift or reduce as desired. YACC support:

```
%left PLUS MINUS
%left STAR SLASH
```

- ✓ Terminals listed in increasing precedence
- ✓ Precedence of a rule is that of its last terminal
- ✓ Reduce if rule precedence greater than next token, or if same and rule precedence is left-associative. Otherwise shift.

Unary minus

Consider

```
-3 * 4 - 6
```

We want these rules to be in increasing precedence:

```
exp: exp MINUS exp
   | exp TIMES exp
   | MINUS exp
```

But YACC would give the same precedence to the first and third rules.

Solution: Imaginary Tokens + Override Rule Precedence

```
%left PLUS MINUS
%left STAR SLASH
%left UNARY_MINUS

exp: exp MINUS exp
   | exp TIMES exp
   | MINUS exp      %prec UNARY_MINUS
```

Solution: Imaginary Tokens + Override Rule Precedence

```
%left PLUS MINUS
%left STAR SLASH
%left UNARY_MINUS

exp: exp MINUS exp
   | exp TIMES exp
   | MINUS exp      %prec UNARY_MINUS
```

The lexer never produces a `UNARY_MINUS` token!

Lexer Generators

Tools like LEX, FLEX, ALEX, MLLEX, JLEX, *etc.*, can take descriptions of the tokens (regular expressions), and generate the necessary code to implement a lexer.
They differ primarily in the language of the generated code.

Input file (LEX/FLEX):

```
 ...definitions and configuration variables...
%%
 ...rules (regular expressions + C code
     to return the run-time token representation)...
%%
 ...C helper functions used for building
    representations...
```

Input file (ALEX):

```
{
 ...Haskell code posted into top of output file
}
 ...definitions and configuration variables...

 ...rules (regular expressions + Haskell code
     to build the run-time token representation)...
{
 ...Haskell helper functions used for building
    representations...
}
```

Describing Tokens

Lex (C)

```
if
then
else
"<"
"<="
[0-9]+
[A-Za-z][A-Za-z0-9_]*
[ \t\n]+
```

Alex (Haskell)

```
if
then
else
"<"
"<="
[0-9]+
[A-Za-z][A-Za-z0-9_]*
[ \t\n]+
```

Describing Tokens

Lex (C)

```
alph  [A-Za-z]
digit [0-9]
ident {alph}({alph}|{digit})*
ws    [ \t\n]+
%%
if
then
else
"<"
"<="
{digit}+
{ident}
\"[^\"]*\"
{ws}
```

Alex (Haskell)

```
$alph  = [A-Za-z]
$digit = [0-9]
@ident = $alph($alph|$digit)*
-- $white is predefined

if
then
else
"<"
"<="
$digit+
@ident
\"[^\"]*\"
$white+
```

LEX: RULES

```
if              { return IF; }
then            { return THEN; }
else            { return ELSE; }
"<"             { return '<'; }
"<="            { return LEQ; }
{digit}+        { yylval.ival = atoi(yytext);
                  return ICONST; }
{ident}         { yylval.sval = strdup(yytext);
                  return IDENT; }
\"[^\"]*\"      { yylval.sval = strdup(yytext);
                  return SCONST; }
{ws}            {}
```

(Note: Assumes `IF`, `THEN`, *etc.*, have been previously defined as integer constants, and that `yylval` is a global `union` variable.)

Lex: Keeping Track of Line Numbers

Some implementation also define `yylineno` in addition to `yytext`

```
{ident}      { printf("DEBUG: variable '%s' on line %d\n",
                       yytext, yylineno);
               return IDENT; }
```

Lex: Keeping Track of Line Numbers

Some implementation also define `yylineno` in addition to `yytext`

```
{ident}       { printf("DEBUG: variable '%s' on line %d\n",
                          yytext, yylineno);
               return IDENT; }
```

If not, we can add this ourselves.

```
%{
int lineno;
%}

%%

[ \t]         {}
\n            { ++lineno; }
{ident}       { printf("DEBUG: variable '%s' on line %d\n",
                          yytext, lineno);
                return IDENT; }
```

ALEX RULES: "BASIC" WRAPPER

```
if            { \yytext -> IF }
then          { \yytext -> THEN }
else          { \yytext -> ELSE }
"<"           { \yytext -> LT }
"<="          { \yytext -> LEQ }
{digit}+      { \yytext -> ICONST (read yytext) }
{ident}       { \yytext -> IDENT (yytext) }
\"[^\"]*\"    { \yytext -> SCONST (init (tail yytext)) }
$white+       ;
```

Note: Assumes we previously defined a type

```
data Token = IF | THEN | ELSE | LT | LEQ
         | ICONST Integer
         | IDENT String | SCONST String
```

ALEX RULES: "POSN" WRAPPER

You can tell ALEX that you want to write actions of the form

```
if          { \pos -> \yytext -> IF pos }
then        { \pos -> \yytext -> THEN pos }
else        { \pos -> \yytext -> ELSE pos }
"<"         { \pos -> \yytext -> LT pos }
"<="        { \pos -> \yytext -> LEQ pos }
{digit}+    { \pos -> \yytext -> ICONST (read yytext) pos }
  ...snip...
$white+     ;
```

Note: Assumes we previously defined a type

```
data Token = IF AlexPosn | ...
           | IDENT String AlexPosn | ...
```

Start States

Most lexing tools also provide multiple "start states."
The current "state" determines which rules apply.
Particularly useful for string constants, long comments, etc.

Flex:

```
%x CMNT
%%
"/*"          { BEGIN CMNT; }

<CMNT>.       {}
<CMNT>\n      { ++lineno }
<CMNT>"*/"    { BEGIN INITIAL; }
<CMNT><<EOF>> { printf("Unclosed comment\n";
               ...snip... }
```

Alex:

```
<0>"/*"        { begin comment }

<comment>.     ;
<comment>\n    ;
<comment>"*/" { begin 0 }
```

- ✓ Warning: some tools say a rule with no state annotation *always* applies.
- ✓ What if we wanted nested comments?

What if whitespace is significant?

```
if (space_left != 0):
    processLine(line)
    getNextLine()
 else:
    getNextLine()
    skipped = 1
```

Yacc, `Bison`, Happy, and other parser generators

Similar ideas as with lexer generators.

- ✓ Specify what the tokens are (plus any associated values)
- ✓ Specify what the grammar is
- ✓ Specify what value to associate with each nonterminal on the stack
  - Code fragment that execute when a rule is reduced
  - They can refer to previously-computed values associated with handle items.
  - Most often, these "values" are abstract syntax trees.

Most tools generate $LR(1)/LALR(1)$ parsers; `Antlr` creates an $LL(1)$ parser.

Example: Pic expressions

```
Exp :: { Absyn.Exp }

Exp: num                    { Num $1 }
   | name                   { Var $1 }
   | '(' Exp ')'            { $2 }
   | '-' Exp %prec NEG      { BinOp (Num 0) Minus $2 }
   | Exp '+' Exp            { BinOp $1 Plus $3 }
   | Exp '-' Exp            { BinOp $1 Minus $3 }
   | Exp '*' Exp            { BinOp $1 Times $3 }
   | Exp '/' Exp            { BinOp $1 Divide $3 }
```

See the handout.

Monads Summarized

Monads are the way that Haskell lets you write side-effecting code.

Key Monad Concepts (1)

For any monad `M`, the type `M a` describes "computations" that, if performed, do stuff and provide a result of type `a`

Key Monad Concepts (1)

For any monad `M`, the type `M a` describes "computations" that, if performed, do stuff and provide a result of type `a`

```
-- IO is a monad

getChar :: IO Char
putChar :: Char -> IO ()
```

Key Monad Concepts (2)

There is a way to combined ("sequence") computations into one big computation.

Key Monad Concepts (2)

There is a way to combined ("sequence") computations into one big computation.

```
doubleEcho :: IO unit
doubleEcho =
   do x <- getChar
      putChar x
      putChar x
```

Key Monad Concepts (3)

You can create a computation that does nothing execept return a fixed value.

Key Monad Concepts (3)

You can create a computation that does nothing execept return a fixed value.

```
doubleEchoGetchar :: IO Char
doubleEchoGetchar =
   do x <- getChar
      putChar x
      putChar x
      return x
```

Key Monad Concepts (4)

There may or may not actually be a way to run the computation!

Key Monad Concepts (4)

There may or may not actually be a way to run the computation!

```
main :: IO ()
```

Key Monad Concepts (4)

There may or may not actually be a way to run the computation!

```
main :: IO ()
```

```
System.IO.Unsafe.unsafePerformIO :: IO a -> a
```

Guidelines for using **do**

do lets you sequence multiple computations.

- ✓ Each line in the **do** is either a computation to be performed (A Haskell expression whose result a computation) or defining a variable as usual in Haskell
- ✓ The last line must be a computation (producing the result of the whole sequence).
- ✓ The computations must all be within the same monad.
- ✓ Computation lines generally contain `... <- ...`
  Regular haskell lines generally need `let ... = ...`

Example: ToJVM.hs

```
main :: IO ()
main =
  do args <- System.getArgs                -- System.getArgs :: IO [String]
     let fileName = case args of
                      [arg] -> arg
                      _ -> error "exactly one filename expected"
     let outFileName = fileName ++ ".j"

     fileContents <- readFile fileName     -- readFile :: String -> IO String
     let tokens = tokenize fileContents
     let ast = parse tokens

     let (numLocalVars, code) = transP ast

     let headerCode = [".class public Main",
                       ".super  java/lang/Object",
                       ".method public static main([Ljava/lang/String;)V",
                       ".limit locals " ++ show numLocalVars,
                       ".limit stack 99"]

     let trailerCode = ["\treturn",
                        ".end method"]

                                       -- writeFile :: String -> String -> IO ()
     writeFile outFileName (unlines (headerCode ++ code ++ trailerCode))
     return ()                         -- Could be omitted. (Why?)
```

Other Monads

- ✓ Lists are monads ("computations that may return any number of values")

Other Monads

- ✓ Lists are monads (“computations that may return any number of values”)

```
do x <- [1,2,3,4]
   y <- [5,6,7]
   return (x*y)
```

Other Monads

- ✓ Lists are monads ("computations that may return any number of values")

```
do x <- [1,2,3,4]
   y <- [5,6,7]
   return (x*y)
```

$\hookrightarrow$ `[5,6,7,10,12,14,15,18,21,20,24,28]`

NB: Without parens, `ghc` sees `(return x) * y`

Other Monads

- ✓ Lists are monads ("computations that may return any number of values")

```
do x <- [1,2,3,4]
   y <- [5,6,7]
   return (x*y)
```

  $\hookrightarrow$ `[5,6,7,10,12,14,15,18,21,20,24,28]`

  NB: Without parens, `ghc` sees `(return x) * y`

- ✓ Maybes are monads ("computations that return at most one value")

Other Monads

- ✓ Lists are monads ("computations that may return any number of values")

```
do x <- [1,2,3,4]
   y <- [5,6,7]
   return (x*y)
```

  $\hookrightarrow$ `[5,6,7,10,12,14,15,18,21,20,24,28]`

  NB: Without parens, **ghc** sees `(return x) * y`

- ✓ Maybes are monads ("computations that return at most one value")
- ✓ Haskell defines a "state monad" **ST** (computations that read and write memory but have no other side-effects)

Other Monads

- ✓ Lists are monads ("computations that may return any number of values")

```
do x <- [1,2,3,4]
   y <- [5,6,7]
   return (x*y)
```

  ↪ `[5,6,7,10,12,14,15,18,21,20,24,28]`

  NB: Without parens, `ghc` sees `(return x) * y`

- ✓ Maybes are monads ("computations that return at most one value")
- ✓ Haskell defines a "state monad" `ST` (computations that read and write memory but have no other side-effects)
- ✓ Can define your own monads
  - Define a type `M a`
  - Define `return :: a -> M a`
  - Define `>>= :: M a -> (a -> M b) -> M b` (composes 2 computations)
  - Optionally, define any built-in computations (e.g., `putChar`)
  - Optionally, define a way to run a computation (`:: M a -> a`)

Example from the Homework

```
compileExp :: Exp -> TrM ([String], String)

compileExp (EBop exp1 PlusOp exp2) =
 do (code1, t1) <- compileExp exp1
    (code2, t2) <- compileExp exp2
    t3 <- freshTemp
    let code3 = [t3 ++ " = add i32 " ++ t1 ++ ", " ++ t2]
    return (code1 ++ code2 ++ code3, t3)
```