Parsing, Concluded

CS 132: Compiler Design

February 2, 2011
Dangling Else

\[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \]
  \[ | \text{if } E \text{ then } S \]
  \[ | ... \]

How to parse

\[ \text{if } a \text{ then if } b \text{ then } s1 \text{ else } s2 \]
Dangling Else

\[ S \rightarrow \text{if } E \text{ then } S \text{ else } S \]
\[ | \text{if } E \text{ then } S \]
\[ | \ldots \]

How to parse

\[ \text{if a then if b then } s_1 \text{ else } s_2 \]

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

\[
S \rightarrow \text{if } E \text{ then } S \text{ . else } S \quad <\text{any}> \\
S \rightarrow \text{if } E \text{ then } S \text{ . else}
\]
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

\[
\begin{align*}
S & \rightarrow M \\
S & \rightarrow U \\
M & \rightarrow \text{if } E \text{ then } M \text{ else } M \\
M & \rightarrow \ldots \\
U & \rightarrow \text{if } E \text{ then } S \\
U & \rightarrow \text{if } E \text{ then } M \text{ else } U
\end{align*}
\]

Hack the language

\[
\begin{align*}
S & \rightarrow \text{if } E \text{ then } S \text{ else } S \text{ endif} \\
& \quad | \text{if } E \text{ then } S \text{ endif} \\
& \quad | \ldots
\end{align*}
\]

Hack the parser:
Whenever you reach this “conflicting” state, just go ahead and shift.
**Ambiguous Expression Grammars**

\[ E \rightarrow E + E \mid E - E \]
\[ \mid E * E \mid E / E \]
\[ \mid (E) \mid n \]
Ambiguous Expression Grammars

\[ E \rightarrow E \cdot + E \mid E \cdot - E \]
\[ \mid E \cdot * E \mid E \cdot / E \]
\[ \mid ( E ) \mid n \]

All sorts of shift-reduce conflicts:

\begin{align*}
E \rightarrow & \quad E \cdot + E \quad \langle \text{any} \rangle \\
E \rightarrow & \quad E \cdot * E \quad + \\
E \rightarrow & \quad E \cdot * E \quad \langle \text{any} \rangle \\
E \rightarrow & \quad E \cdot + E \quad * \\
E \rightarrow & \quad E \cdot + E \quad \langle \text{any} \rangle \\
E \rightarrow & \quad E \cdot + E \quad + \\
\end{align*}
Solutions

Hack the grammar

\[
\begin{align*}
E & \rightarrow E + T \mid E - T \mid T \\
T & \rightarrow T \ast F \mid T / F \mid F \\
F & \rightarrow (E) \mid n
\end{align*}
\]

\[
\begin{align*}
E & \rightarrow T (+ T \mid - T)^
\end{align*}
\]

\[
\begin{align*}
T & \rightarrow F (* F \mid / F)^
\end{align*}
\]

\[
\begin{align*}
F & \rightarrow (E) \mid n
\end{align*}
\]
**Solutions**

Hack the grammar

\[
E \rightarrow E + T \mid E - T \mid T \\
T \rightarrow T * F \mid T / F \mid F \\
F \rightarrow (E) \mid n
\]

\[
E \rightarrow T (+ T \mid - T)^* \\
T \rightarrow F (* F \mid / F)^* \\
F \rightarrow (E) \mid n
\]

Hack the language

\[
E \rightarrow + E E \mid * E E \mid \ldots
\]
Solutions

Hack the grammar

\[
E \rightarrow E + T | E - T | T \\
T \rightarrow T * F | T / F | F \\
F \rightarrow ( E ) | n
\]

\[
E \rightarrow T (+ T | - T)^* \\
T \rightarrow F (* F | / F)^* \\
F \rightarrow ( E ) | n
\]

Hack the language

\[
E \rightarrow + E E | * E E | \ldots
\]

Hack the parser to shift or reduce as desired.

\[
E \rightarrow E . + E <\text{any}> \\
E \rightarrow E * E . + \\
E \rightarrow E . * E <\text{any}> \\
E \rightarrow E + E . * \\
E \rightarrow E . + E <\text{any}> \\
E \rightarrow E + E . +
\]
Solutions

Hack the grammar

\[
E \rightarrow E + T \mid E - T \mid T \\
T \rightarrow T * F \mid T / F \mid F \\
F \rightarrow ( E ) \mid n
\]

\[
E \rightarrow T \left( + T \mid - T \right)^* \\
T \rightarrow F \left( * F \mid / F \right)^* \\
F \rightarrow ( E ) \mid n
\]

Hack the language

\[
E \rightarrow + E E \mid * E E \mid \ldots
\]

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

\[
\%left PLUS MINUS \\
\%left STAR SLASH
\]
Solutions

Hack the grammar

\[
E \rightarrow E + T \mid E - T \mid T \\
T \rightarrow T * F \mid T / F \mid F \\
F \rightarrow (E) \mid n
\]

\[
E \rightarrow T (+ T \mid - T)* \\
T \rightarrow F (* F \mid / F)* \\
F \rightarrow (E) \mid n
\]

Hack the language

\[
E \rightarrow + E E \mid * E E \mid ...
\]

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

\[
\begin{align*}
E &\rightarrow E + T \mid E - T \mid T \\
T &\rightarrow T * F \mid T / F \mid F \\
F &\rightarrow ( E ) \mid n
\end{align*}
\]

\[
\begin{align*}
E &\rightarrow T (+ T \mid - T)^* \\
T &\rightarrow F (* F \mid / F)^* \\
F &\rightarrow ( E ) \mid n
\end{align*}
\]

Hack the language

\[
E \rightarrow + E E \mid * E E \mid \ldots
\]

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

\[
\begin{align*}
&\%\text{left PLUS MINUS} \\
&\%\text{left STAR SLASH}
\end{align*}
\]

✓ 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

\[-3 \times 4 - 6\]

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, aplex, 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)**

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

**Alex (Haskell)**

```plaintext
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]+

```plaintext
%%
if
then
else
"<"
"<="
{digit}+
{ident}
"\^[^\"]*" {ws}
```

**Alex (Haskell)**

```plaintext
$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`

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

If not, we can add this ourselves.

```c
{ %
  int lineno;
%
  { ++lineno; }
  {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`

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

If not, we can add this ourselves.

```c
{%
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:

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

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

Alex:

```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?

```python
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; \( \texttt{Antlr} \) 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 }
<table>
<thead>
<tr>
<th>Practical Parsing Issues</th>
<th>Lexer Generators</th>
<th>Parser Generators</th>
<th>Monads</th>
</tr>
</thead>
</table>


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 $a$

```haskell
-- 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.

```haskell
doubleEcho :: IO unit
doubleEcho =
  do x <- getChar
     putChar x
     putChar x
```
**Key Monad Concepts (3)**

You can create a computation that does nothing except return a fixed value.
**Key Monad Concepts (3)**

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

```haskell
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!

```haskell
main :: IO ()
```
Key Monad Concepts (4)

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

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

```haskell
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**

```haskell
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 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”)

```haskell
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
Other Monads

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

```haskell
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`
Other Monads

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

```haskell
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”)

▶ Deène a type `M a`
▶ Deène `return :: a -> M a`
▶ Deène `>>= :: M a -> (a -> M b) -> M b` (composes computations)
▶ Optionally, deène any built-in computations (e.g., `putChar`)
▶ Optionally, deène away to run a computation (`:: M a -> a`)
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)
Other Monads

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

\[
\begin{align*}
\text{do } & \ x \leftarrow \ [1,2,3,4] \\
\text{do } & \ y \leftarrow \ [5,6,7] \\
\text{return} & \ (x\cdot y) \\
\rightarrow & \ [5,6,7,10,12,14,15,18,21,20,24,28]
\end{align*}
\]

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

✓ Maybes are monads (“computations that return at most one value”)

✓ Haskell defines a “state monad” \(\text{ST}\) (computations that read and write memory but have no other side-effects)

✓ Can define your own monads

- Define a type \(\text{M } a\)
- Define \(\text{return} :: a \rightarrow \text{M } a\)
- Define \(\text{>>=} :: \text{M } a \rightarrow (a \rightarrow \text{M } b) \rightarrow \text{M } b\) (composes 2 computations)
- Optionally, define any built-in computations (e.g., `putChar`)
- Optionally, define a way to run a computation (\(:: \text{M } a \rightarrow a\))
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)
```