Applying ML-Yacc

February 5, 2001
CS 132: Compiler Design

Review: Arithmetic Precedence

• Naive expression grammar generates shift-reduce conflicts, including

\[
\begin{align*}
E &\rightarrow E + E, \text{any} \\
E &\rightarrow E * E, + \\
E &\rightarrow E * E, \text{any} \\
E &\rightarrow E + E, * \\
E &\rightarrow E - E, \text{any} \\
E &\rightarrow E - E, - \\
\end{align*}
\]
Review: Arithmetic Precedence

• In the case

\[
\begin{align*}
E & \rightarrow E \cdot E \cdot + \cdot \text{any} \\
E & \rightarrow E \cdot E \cdot + \\
E & \rightarrow E \cdot E \cdot + \\
E & \rightarrow E \cdot + \cdot \text{any} \\
E & \rightarrow E \cdot + \\
\end{align*}
\]

we have seen \(E \cdot E\) and the lookahead is +

• We can either shift the + onto the stack or reduce via \(E \rightarrow E \cdot E\).

• Since we want multiplication to group more strongly than addition, correct answer is \textit{reduce}.

Review: Arithmetic Precedence

• In the case

\[
\begin{align*}
E & \rightarrow E \cdot * \cdot E \cdot \text{any} \\
E & \rightarrow E \cdot * \\
\end{align*}
\]

we have seen \(E + E\) and the lookahead is *

• We can either shift the * onto the stack or reduce via \(E \rightarrow E + E\).

• Since we want multiplication to group more strongly than addition, correct answer is \textit{shift}. 
Review: Arithmetic Precedence

• In the case

\[
\begin{align*}
[E \rightarrow E \cdot E, \text{any}] \\
[E \rightarrow E \cdot E, , -]
\end{align*}
\]

we have seen $E - E$ and the lookahead is $-$

• We can either shift the $-$ onto the stack or reduce via $E \rightarrow E - E$.

• Since we want subtraction to be left-associative, correct answer is \textit{reduce}.

Resolving Shift/Reduce Conflicts

• ML-Yacc permits terminals to be given a precedence and associativity.

\[
\begin{align*}
\%\text{nonassoc EQ} \\
\%\text{left PLUS MINUS} \\
\%\text{right TIMES}
\end{align*}
\]

• Each rule may have a precedence and associativity.
  - Defaults to that of rightmost terminal
  - Can be overridden by \%\text{prec} specifier
    • Recall the unary-minus example
Resolving Shift/Reduce Conflicts

- Conflict resolution
  - Assume choices are
    - shift a terminal \( t \)
    - reduce a rule \( A \rightarrow \beta \)
  - Yacc will reduce
    - if rule's precedence > terminal's precedence
    - Or, if precedences are equal and both are left-associative
  - Otherwise shift.
    - Also shift if at least one missing a specified precedence
    - (Yields original rule of default-to-shift.)

- Consider the previous cases again.

Another Example: Dangling Else

- Keywords then and else may need precedences when we have, e.g., arithmetic operators around.
- Consider
  
  \[
  \text{if 1 then 2 + 4}
  \]

  \[
  \text{if 1 then 2 else 3 + 4}
  \]

- Solution: give then and else low precedence.
- What goes wrong if you make them left-associative, though?
Semantic Actions

• Many terminals carry semantic information
  - e.g., INT terminal having a associated integer
  - e.g., STRING terminal with associated string
• Can generalize this to having information associated with each node in the parse tree.
  - *Semantic actions* compute information for nonterminals, in terms of the values of their children.

Semantic Actions in ML-Yacc

• ML-Yacc associates an arbitrary piece of SML code with each production rule.
  - Stack now contains not symbols, but (symbol,value) pairs.
    • Well, technically there are two parallel stacks.
  - When a reduction rule $A \rightarrow \beta$ applies, runs the corresponding semantic action to get the value for $A$.
    • Usually computed in terms of values for the symbols in $\beta$.
  - LR parsing performs reductions in predictable order
    • As if values were computed in postorder parse tree traversal, although full parse tree may not actually be constructed.
Example: Calculator

```%
%term INT of int | PLUS | MINUS | TIMES | UMINUS | EOF
%nonterm exp of int
%start exp
%
exp : INT (INT)
    | exp PLUS exp (exp1 + exp2)
    | exp MINUS exp (exp1 - exp2)
    | exp TIMES exp (exp1 * exp2)
    | MINUS exp %prec UMINUS (~ exp)
```

Example: RPN Generator

```%
%term INT of int | PLUS | MINUS | TIMES | UMINUS | EOF
%nonterm exp
%start exp
%
exp : INT (emit(Push INT))
    | exp PLUS exp (emit ADD)
    | exp MINUS exp (emit SUB)
    | exp TIMES exp (emit MULT)
    | MINUS exp %prec UMINUS (emit NEG)
```
Example: PN Generator

%%
%term INT of int | PLUS | MINUS | TIMES | UMINUS | EOF
%nonterm exp of instruction list
%start exp
%%
exp : INT     {[Num(INT)]}
| exp PLUS exp {[Add] @ exp1 @ exp2}
| exp MINUS exp {[Sub] @ exp1 @ exp2}
| exp TIMES exp {[Mult] @ exp1 @ exp2}
| MINUS exp %prec UMINUS  {[Neg] @ exp}

Abstract Syntax

• Another use of semantics actions is to build an abstract syntax tree

• Recall:
  - *Concrete syntax* refers to the parse tree generated corresponding to the grammar.
  - *Abstract syntax* retains only the essential information from the parse tree
Example

- Consider the following grammar:

```
<exp> ::= <exp> + <term>  
  | <exp> - <term>  
  | <term>  
<term> ::= <term> * <factor>  
  | <factor>  
<factor> ::= ( <exp> )  
  | <num>
```

Concrete & Abstract Syntax

```
2 - (3 + 5)
```

2 - (3 + 5)
Concrete Syntax

- Concrete syntax is comparatively "arbitrary"

```
<num> ::= one | two | three | ...
<exp> ::= <num>
    | add <exp> and <exp>
    | subtract <exp> from <exp>
    | multiply <exp> by <exp>
    | ( exp )
```

subtract (add three plus five)
from two

review: Concrete Syntax

- Concrete syntax is comparatively "arbitrary"

```
<num> ::= 1 | 2 | 3 | ...
<exp> ::= <num>
    | <exp> <exp> +
    | <exp> <exp> -
    | <exp> <exp> *
```

2 3 5 + -
Abstract Syntax

- Abstract syntax can be specified with a grammar as well
  - Doesn't matter if the grammar is ambiguous
  - We always have a tree, not just a string

```
n ::= 1 | 2 | 3 | ...
```

```
e ::= n | e + e
    | e - e | e * e
```

```
2 - (3+5)
```

Straight-line Abstract Syntax

```
datatype binop = ... and stm = ... and exp = ...

%term INT of int | ID of string | ...
\nonterm exp of exp | stm of stm | exps of exp list
%start stm

stm : stm SEMICOLON stm
    | ID ASSIGN exp
    | PRINT LPAREN exps RPAREN
exps: exp
    | exp COMMA exps
exp : INT
    | ID
    | exp PLUS exp
    | stm COMMA exp
    | LPAREN exp RPAREN
```
Symbols (Hashable Strings)

```ocaml
signature SYMBOL =
  sig
    eqtype symbol
    val symbol : string -> symbol
    val name : symbol -> string
  end

type 'a table
val empty : 'a table
val enter : 'a table * symbol * 'a -> 'a table
val look : 'a table * symbol -> 'a option
end
```

Tiger Abstract Syntax: L-values

```ocaml
type pos = int
and symbol = Symbol.symbol

datatype var = SimpleVar of symbol * pos
  | FieldVar of var * symbol * pos
  | SubscriptVar of var * exp * pos

```
Tiger Abstract Syntax: Exps

```
and exp = VarExp of var
| NilExp
| IntExp of int
| StringExp of string * pos
| CallExp of {func: symbol, args: exp list, pos: pos}
| OpExp of {left: exp, oper: oper, right: exp, pos: pos}
| RecordExp of {fields: (symbol * exp * pos) list,
    typ: symbol, pos: pos}
| SeqExp of (exp * pos) list
| AssignExp of {var: var, exp: exp, pos: pos}
| IfExp of {test: exp, then’: exp, else’: exp option,
    pos: pos}
| WhileExp of {test: exp, body: exp, pos: pos}
| ForExp of {var: symbol, escape: bool ref,
    lo: exp, hi: exp, body: exp, pos: pos}
| BreakExp of pos
| LetExp of {decs: dec list, body: exp, pos: pos}
| ArrayExp of {typ: symbol, size: exp, init: exp, pos: pos}
```

Tiger Abstract Syntax: Decs

```
and dec = FunctionDec of fundec list
| VarDec of {name : symbol,
    escape: bool ref,
    typ : (symbol * pos) option,
    init : exp,
    pos : pos}
| TypeDec of {name:symbol, ty:ty, pos:pos} list

withtype fundec = {name : symbol,
    params: field list,
    result: (symbol * pos) option,
    body : exp,
    pos : pos}

and field = {name : symbol, escape: bool ref,
    typ : symbol, pos : pos}
```
Tiger Abstract Syntax: Types

\[
\begin{align*}
\text{and } & \quad \text{ty} = \text{NameTy of symbol } \ast \text{ pos} \\
& \quad | \quad \text{RecordTy of field list} \\
& \quad | \quad \text{ArrayTy of symbol } \ast \text{ pos} \\
\text{withtype } & \quad \text{field} = \{ \text{name} : \text{symbol}, \\
& \quad \quad \text{escape: } \text{bool ref}, \\
& \quad \quad \text{typ} : \text{symbol}, \\
& \quad \quad \text{pos} : \text{pos} \}
\end{align*}
\]

Example

- The Tiger program
  
  \[(a := 5; a+1)\]

  translates to

\[
\text{SeqExp}\{(\text{AssignExp}\{\text{var} = \text{SimpleVar}(\text{symbol } "a", 2), \\
\quad \text{exp} = \text{IntExp} 5, \\
\quad \text{pos} = 4\}, 2), \\
\quad (\text{OpExp}\{\text{left} = \text{VarExp}(\text{SimpleVar}(\text{symbol } "a", 10)), \\
\quad \text{oper} = \text{PlusOp}, \\
\quad \text{right} = \text{IntExp} 1, \\
\quad \text{pos} = 11\}, 10)\}\
\]
Positions

• In semantic actions, the ML code can get not just values associated with tokens, but also their start/end positions in the source code
  - Instead of "exp" or "exp1" or "exp2"
  - Write "expleft" "expright", "exp1left", "exp1right", "exp2left", "exp2right"

• The abstract syntax tree contains only a single character position for each node
  - Choosing this is a "matter of taste"

A Final Digression

• In the SML language one can dynamically make operators infix:

```
fun plus(n:int,m:int) = n+m
val x = plus (3,4)

infix 4 plus
val y = 1 plus 2
```

• How is this handled in ML-Yacc?