

CS134b: Lexing and Parsing

- Java Parsing
 - Common Problems
- Style
 - Syntax
 - Semantics



Left vs. right recursion in yacc

- Right recursion is “bad”
 - Uses up more stack space in the PDA
- Left recursion is “good”
 - We could consider this strange in a LR grammar...
- But, right recursion seems to be easier in ML:
 - `args: exp`
 - `{ $1 }`
 - `| exp TokComma args`
 - `{ $1 :: $3 }`
 - `;`
- This is bad:
 - `args: exp { $1 } | args TokComma exp { $1 @ [$3] } ;`



Parsing: simple sequence parsing

How to parse:

`prod ::= (a1 a2 ... an)*`

```
prod:
  /* empty */
  { [] }
  | rev_prod_list
  { List.rev $1 }
  ;
```

```
rev_prod_list:
  prod
  { $1 }
  | rev_prod_list prod
  { $2 :: $1 }
  ;

prod:
  a1 a2 ... an
  { make_prod $1 $2 ... $n }
  ;
```



Separated sequence parsing

How to parse:

args ::= exp | args , exp

```
args:
  /* empty */
  { [] }
| rev_args
  { List.rev $1 }
;
```

```
rev_args:
  exp
  { [$1] }
| rev_args TokComma exp
  { $3 :: $1 }
;
```



Expressions: shift/reduce conflicts

- This grammar is totally ambiguous

```
- exp:
  TokInt
  { IntExpr $1 }
| TokId
  { VarExpr $1 }
| ...
| exp TokPlus exp
  { make_binop PlusOp $1 $3 }
| exp TokStar exp
  { make_binop MulOp $1 $3 }
| ...
;
```



Precedences

- Just add precedence declarations

```
- ...
- %left TokLsl TokLsr TokAsr
- %left TokPlus TokMinus
- %left TokStar TokSlash TokPercent
- ...
```



Problem #3

- Declarations in Java/C are bogus
- What is the type of `int[] f()(int i) (int j, int k);`
 - Is it a function? Is it an array? Its two taste treats in one!
- Java/C declarations are parsed inside-out, but interpreted outside-in
 - The type of `f` in ML would be:
 - `(int -> ((int * int -> int array) array) array)`
- What to do? When we encounter an identifier, we have NO idea what its type should be...



Solution #3: use a temp type

- This represents the parsed identifier (outside-in)
 - type `var_name =`
 - `VarNameId of symbol * pos`
 - `| VarNameArray of var_name * pos`
 - `| VarNameFun of var_name * (symbol * ty) list * pos`
- `Direct_decl`
 - `Direct_decl:`
 - `TokId`
 - `{ let id, pos = VarNameId (id, pos) }`
 - `| direct_decl TokDoubleBrack`
 - `{ VarNameArray ($1, ...) }`
 - `| direct_decl TokLeftParen opt_var_decl_list TokRightParen`
 - `{ VarNameFun ($1, make_param_decls $3, ...) }`



Temp type is backward!

- Reverse it in the topmost semantic action
- Build a function that returns: a) the variable, b) its type, and c) its position:
 - `let rec make_var_decl ty = function`
 - `VarNameId (n, pos) ->`
 - `n, ty, pos`
 - `| VarNameArray (v, pos) ->`
 - `make_var_decl (TypeArray (ty, pos)) v`
 - `| VarNameFun (v, args, pos) ->`
 - `make_var_decl (TypeFun (List.map snd args, ty, pos)) v`



Info #4:

- Position info
 - For each expression and type, we want the position
 - In the next phase, we'll report the position if a type-check fails
- Position is a 5-tuple
 - Filename, start_line, start_char, end_line, end_char
- The `get_lexeme` function computes the position of the token by counting the number of `\n` in the string
- During parsing, compute the position by taking the union of all positions
 - `Exp :: exp TokPlus exp`
 - `{let pos1 = pos_of_expr $1 in`
 - `let pos2 = pos_of_expr $3 in`
 - `let pos = union_pos pos1 pos2 in`
 - `BinopExpr (PlusOp, e1, e2, pos)`
 - `}`



Conflict #5

- Constructors vs. methods
- Class X {
 - `int i;`
 - `int f(int i) { return i; }`
 - `X(int i) { this.i = i; }`
- }
- Grammar:
 - `Type_spec: TokId | type_spec TokDoubleBrack`
 - `Fun_def: type_spec direct_decl TokLeftCurly stmt_list TokRightCurly`
 - `const_def: direct_decl TokLeftCurly stmt_list TokRightCurly`



Idea #5

- The problem is that `type_spec` can start with `<id>[]*` and so can `direct_decl`
- Solution:
 - Separate `id_brackets` into a separate production
 - `Id_brackets: TokId | id_brackets TokDoubleBrack`
 - `Type_spec: id_brackets`
 - `Fun_decl: id_brackets TokLeftParen ... TokRightParen`
 - | `fun_decl TokDoubleBrack`
 - | `fun_decl TokLeftParen ... TokRightParen`
 - `Direct_decl: id_brackets | fun_decl`



Problem #6, all those parens!

- The paren forms:
 - `exp: exp TokLeftParen args TokRightParen`
| `TokLeftParen exp TokRightParen`
| `TokLeftParen type_spec TokRightParen exp`
- Several ways to solve, one way is to break expr up into two levels.
 - `Exp: exp2 | exp1`
`exp2: TokInt | exp + exp | ...`
`exp1: TokId | exp1 . TokId | exp1 [exp] | exp1 (args)`
| `(TokId) | (exp2)`
| `(TokId) exp1 | (TokId brackets) | exp1`


