**Final**

CS4410 Compilers: Assignment 2 Writeup

Partners: Kevin Luo, Brian Roels

Shift-Reduce Conflict Resolutions:

First, we used a hierarchy of precedence rules to aid in shift-reduce conflict resolution. In the order of lowest to highest precedence:

* ASSIGN, and OF all bind loosest to ensure proper nesting behavior with assignments and array initialization
* ELSE binds tighter than THEN so a dangling ELSE applies to the closest IF
* ELSE also binds looser than operators so that we can return a calculated value.
* All comparison operators, first and & or then equals, not equals, less than, greater than and the like bind slightly tighter
* All arithmetic operators bind tighter, times and divide binding tighter than add and subtract (via order of operations)
* For declarations, we have a dummy terminal DECS that binds looser than than TYPE and FUNCTION. This is to ensure that when we build lists of mutually recursive functions / types, we actually store them as lists
* DO binds tighter than all operators, although we don’t think this matters, as any operator operating on a while loop or its body will cause a type error later (neither is allowed to produce a value)

We also had to be careful when determining the grammar of any sequence of expressions, specifically type fields, expression sequences, record fields, expression lists, type declaration lists, and function declaration lists.

* Type fields, expression sequences, record fields, and expression lists are all comma-delimited so we needed a number of cases to ensure we could distinguish an empty list from a one element list with no commas to a 2+ element list with n-1 commas. Generally we were able to employ 2 states: 1 as the ‘entrypoint’ for the sequence with an option for the empty sequence and another for the one element and rest, then in parsing the rest a similar but recursive state that includes the shifting of the delimiter.
* Both forms of declaration lists were simpler cases. As there is no delimiter in this syntax, we were able to leverage just one recursive statej.
* Lastly there is the sequence expression case which since it requires 2 or more expression elements was as simple as a single, recursive state and semantic action

As we were careful to parse sequences using left-recursion, many sequences are actually parsed in reverse. In these scenarios such as all declarations this order does not actually matter, any recursive declarations will be handled the same by other components of the parser. Where it does matter in the case of a sequence expression that needs to adhere to the type off the return of the final sequence, we simply reversed this list post-construction before we finalized it as the corresponding abstract syntax form for the sequence expression.

Other Areas of Interest:

* We extended our error message module to support line/column position errors reported by the parser. For the lexer we could rely on the current position of the token being from the line whose position is at the top of the stack, but with parser errors we’ve already compiled all line positions and numbers and so for an accurate file position we need to do some looking back into the stack of line positions to determine the appropriate offset.
* We use the empty expression sequence to indicate when there is no value to return.
* The case of ID LBRACK is ambiguous, since we don’t know if it is the beginning of an lvalue with an index access or a array initialization. We solve this by looking ahead until we figure out the rest of the statement and then applying the proper constructions to the initial value. In the case of the lvalue, we have semantic actions that produce lambdas so that our list, which gets accumulated in reverse order, puts the initial ID in the innermost VarExp.

**Raw**

Since LR parsers use constant stack when recursion in rules is left, we do that whenever possible, like the arguments to a function, a sequence of expressions, the fields of a record, etc  
  
some shift-reduce conflicts we resovled and how: multiplication and division bind tighter than addition and subtraction. All all left-associative, although this only matters for subtraction and division.  
  
Relative operators, <, > and the like ,bind tighter than boolean operators, but not as tightly as arith ops  
  
assignment binds loosest than all of the above.  
  
DO binds tighter than operators, since a while loop's body does not produce a value, although it seems that calling an operator an this value will cause problems during typechecking anyway  
  
operators bind more tightly than then's , since we assume that we want to return the sum if the atatement is true. This one is not so cut and dry ,b ut we think this is the right decision.  
  
we resolve the if then elses to the innermost statement by making then of higher precedence then else  
  
when creating an array, we make operators of higher precedence on the chain, so that the calculated value is put into the array.  
  
we use the precedence rules for each operator when reducing expressions with operators.  
  
we have separate cases for the empty record type and value, which is different from the empt record type value  
  
lists of tydecs, fundecs, record fields and args are backwards, because it doesn't matter  
  
we have face terminal for preceence for tydec, fundecn and decs  
  
accessor lambda thing  
  
we used the empt sequence for no value expressions