We are planning on making a compiler in Eli that takes Mystery code and creates SRC assembly. For Mystery, we are going to use the attached syntax, and the 9-17-2004 version of the PLDetective to determine the correct behavior, given the choices from the table on the last page. For SRC, we are taking the syntax from the Second Edition of Computer Systems Design and Architecture, by Vincent P. Heuring and Harry F. Jordan. We are planning on using an emulator to run the SRC assembly code.

For Mystery, we have decided to support all of the syntax of the language, but only to support only one of the options for each choice presented in PLDetective for the semantics. The semantics we chose are generally the most usable for writing programs in Mystery, such as allowing both widening and narrowing conversions for variables.

For name analysis, Mystery has some demanding requirements with the ability to define procedures that have local variables that aren’t accessible to the rest of the program. We are not going to do dynamic scoping, so the scope for variables can be determined at compile time, and we don’t need to use runtime scope analysis.

We have type analysis as a requirement, because Mystery has subrange types. In edition, runtime bounds checking is necessary, with an appropriate error when the variable is outside the bounds. The PLDetective terminates when there is a bounds error, so we will imitate this behavior. For example, this error looks like: “Error at line 3: Cannot assign 6 to subrange 1..5”

For structured output, we are initially going to simply output from the PTG nodes. The output from the PTG nodes will be SRC assembly, such that the output can be run in a simulator to get the desired output.
Syntax for Mystery

<Program>  ->  <Block> | <Block> ;
<DeclList>  ->  <Decl> | <Decl> ; <DeclList> | e
<Decl>  ->  VAR id : <Type> | TYPE id = <Type> | <ProcDecl>
 ProcDecl  ->  PROCEDURE id ( <Formals> ) : <Type> = <Block>
                  | PROCEDURE id ( <Formals> ) = <Block>
<Formals>  ->  <Formallist> | e
<Formallist>  ->  <Formal> | <Formallist> ; <Formal>
/Formal>  ->  id : <Type>
<Type>  ->  INTEGER | <SubrTy> | <ArrayTy> | id | <ProcTy>
/SubrTy>  ->  [ Number TO Number]
<ArrayTy>  ->  ARRAY <SubrTy> OF <Type>
/ProcTy>  ->  PROCEDURE ( <Formals> ) : <Type> | PROCEDURE
                  ( <Formals> )
<Block>  ->  <DeclList> BEGIN <StmtList> END
<StmtList>  ->  <Stmt> | <Stmt> ; <StmtList> | e
<Stmt>  ->  <Assignment> | <Return> | <Block> | <Conditional>
                  | <Iteration> | <Output> | <Expr>
<Assignment>  ->  <Expr> := <Expr>
<Return>  ->  RETURN <Expr>
<Conditional>  ->  IF <Expr> THEN <StmtList> ELSE <StmtList> END
<Iteration>  ->  WHILE <Expr> DO <StmtList> END
<Output>  ->  PRINT <Expr>
<Expr>  ->  <Operand> | <Expr> <Operator> <Operand>
/Operand>  ->  Number | id | <Operand> [ <Expr> ]
                  | <Operand>( <Actuals> ) | ( <Expr> )
/Operator>  ->  + | > | AND
<Actuals>  ->  <ActualList> | e
<ActualList>  ->  <Expr> | <Actuals> , <Expr>
<table>
<thead>
<tr>
<th>Option</th>
<th>Our Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scoping</td>
<td>Static</td>
</tr>
<tr>
<td>Type of</td>
<td>Numeric have type subrange</td>
</tr>
<tr>
<td>Type equality</td>
<td>Combined (C semantics)</td>
</tr>
<tr>
<td>Type assignability</td>
<td>Narrow or Widen</td>
</tr>
<tr>
<td>Type passibility</td>
<td>Narrow or Widen</td>
</tr>
<tr>
<td>Type operand</td>
<td>Widen</td>
</tr>
<tr>
<td>Type subscript</td>
<td>Narrow or Widen</td>
</tr>
<tr>
<td>Parameter passing</td>
<td>Value</td>
</tr>
<tr>
<td>Argument evaluation order</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>Logical short circuit</td>
<td>Short circuit</td>
</tr>
</tbody>
</table>

Figure 1: Semantic Choices