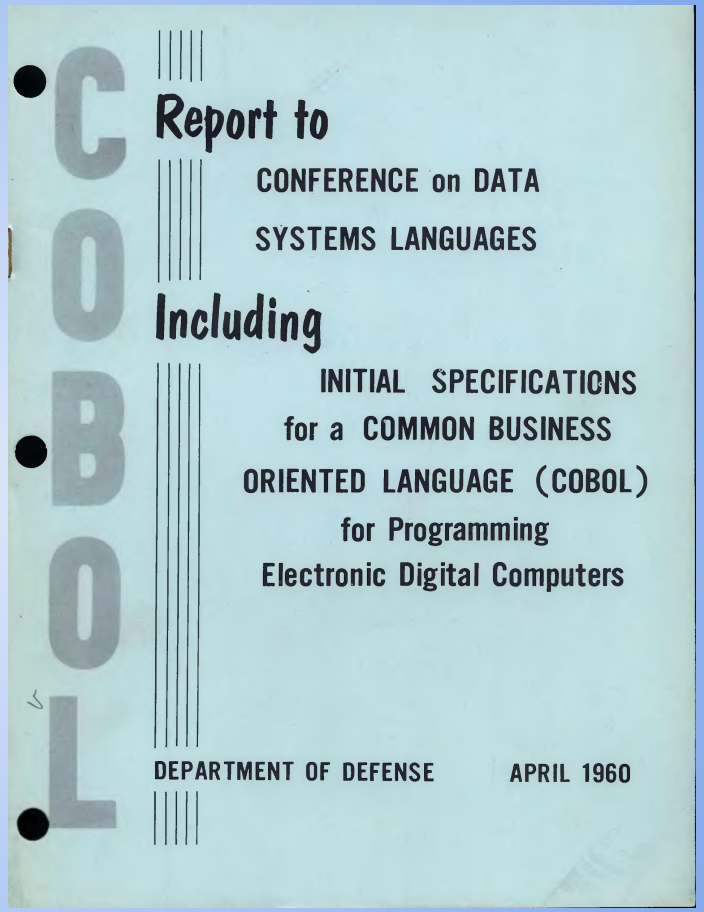
# Compilers

Whiteboard Snapshots
2020 Fall



# – NOTE – NOTE – NOTE – NOTE – NOTE – NOTE –

These pages are just snapshots of ***some*** of the stuff that's scribbled on the "whiteboard" (i.e., those blank pages in the Canvas conference). The ***intent*** is to make copies of whatever goes by.

***However*** …

① There's ***no*** guarantee that there's a copy of everything. Sometimes the discussion is so fast that some pages will slip by unsnapped. ***Take your own notes in class***; let me know if something is missing or should be added.

② I sometimes clean up the snapshots (that's why you'll see typeset pages intermixed with scribbles), there's no guarantee that any particular page will get typeset.

③ There are ***certainly*** tpyos (and even outright *mystekas* — *gasp!*) here. I fix those as I see them, but errors no doubt exist.

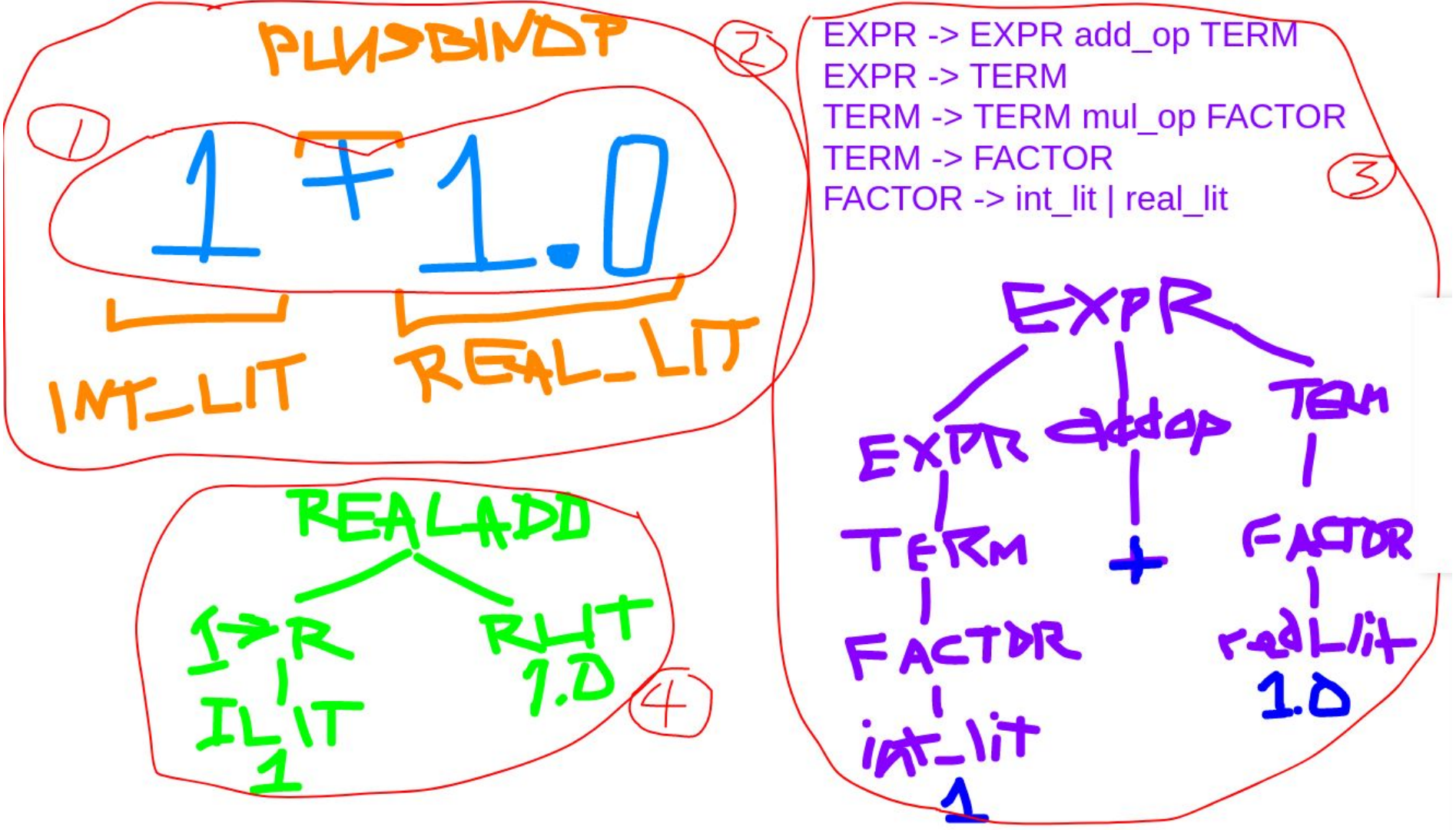
Be mindful of all of this as you review these pages. Tell me if anything is wrong. Thanks!

# SEMANTIC = PROCESSING

Lexical processing -- we used FLEX to transform Regular Expressions to a scanner.
Regular Expressions are a formal notation that FLEX can process in a mechanical way. Characters became tokens.

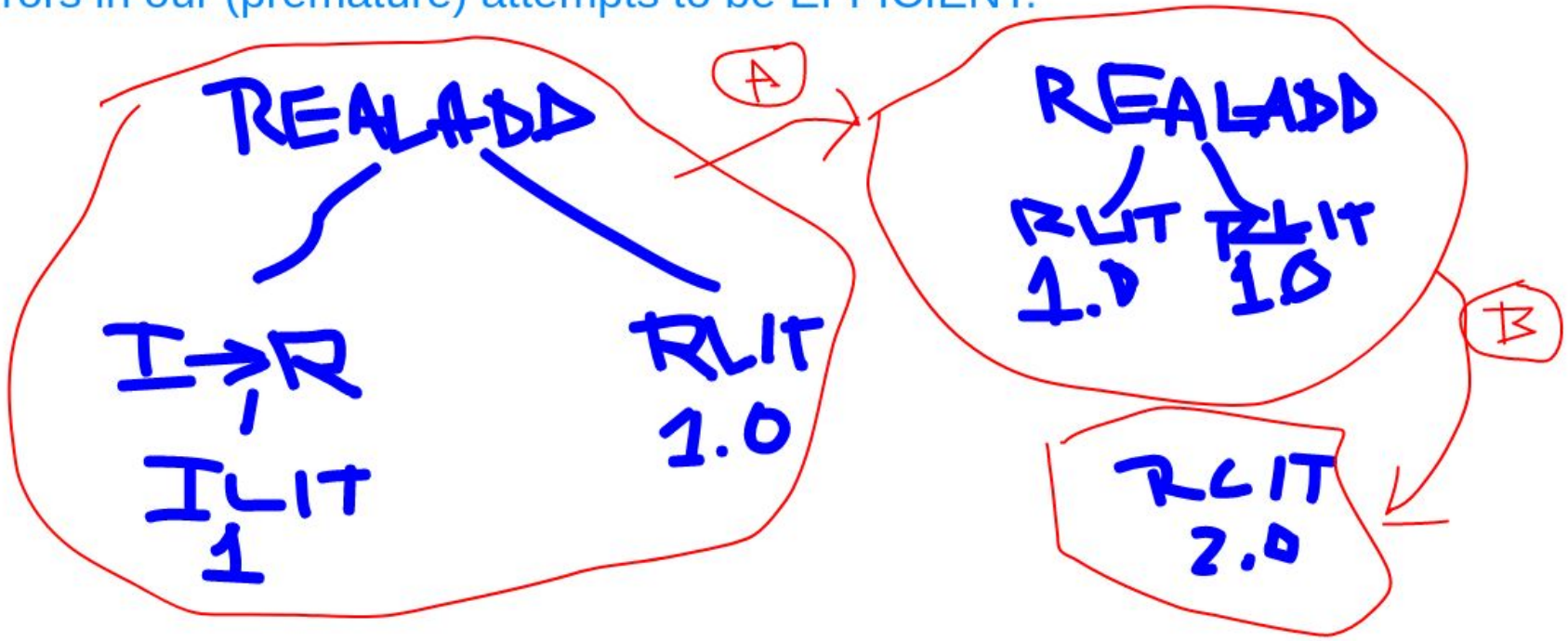
Syntactic processing -- we used BISON to transform a CFG to a parser.
A CFG is a formal notation that BISON can process in a mechanical way.
Tokens became a parse tree -- a Concrete Syntax Tree. (Concrete because the leaves are (usually) the tokens from the input stream.)

Sematic processing -- we will use OUR BRAINS to transform OUR THOUGHTS to a SET OF TREE PROCESSING ROUTINES. The Concrete Syntax Tree becomes an (annotated) Abstract Syntax Tree.


PLUSBINOP
1
1 + 1.0
2
INT_LIT
REAL_LIT
EXPR -> EXPR add_op TERM
EXPR -> TERM
TERM -> TERM mul_op FACTOR
TERM -> FACTOR
FACTOR -> int_lit | real_lit
3
EXPR
EXPR
addop
TERM
TERM
+
FACTOR
FACTOR
real_lit
int_lit
1.0
1
REALADD
REAL
REAL_LIT
INT_LIT
1.0
1
4

The most important characteristic of each step is that it produces the CORRECT next representation. As a consequence, we make the minimal step each time that moves us forward.

We leave OPTIMIZATION as a later step to minimize the chance that we introduce errors in our (premature) attempts to be EFFICIENT.




(Constant Propagation)

If we are performing the Four Basic math ops on arguments, they must be of "suitable" types. "Suitable" depends on the OP itself.

We can add, subtract, multiply, and divide only when the left and right are the "same".

C uses the notion of "promotion". Two ints are the same and an integer operation is computed. If one side is a double, the other side gets "promoted" to double. (Floats are promoted to double for math.)

So if one does int + float, this becomes (double) int + (double) float.

'a' is IDENTICAL to writing 97.

(Technically, depends on your "locale".)

```
char x;  // x is an 8-bit wide int.
short y;  // y is a 16-bit wide int.
int z;      // z is a 32-bit wide int.
long w;  // w is a 64-bit wide int.
```

(Usually. Depends on your installation. On "smaller" architectures, long might be 32 bits instead of 64. Use sizeof() to figure this out.)

"Strings" in C are actually "pointer to char" which is actually "pointer to 8-bit int".

In a CFG ....

function -> type id '(' arglist ')'
'{' statementlist '}'

statementlist -> statementlist statement

statement -> if-statement | for-statement |
expression-statement | while-statement
| return-statement

statementlist -> <empty>

This CFG does NOT require that the func body have at least one return statement.

How could it be rewritten to REQUIRE that there be at least one return -- which is NOT required to be the last statement in the body?

```
fro ( int i=0; i<10; i++ ) {
    printf( "%d\n", i );
}
```

The misspelled keyword ("fro" for "for") is going to be a SYNTACTIC error.

---

Dynamic semantic rules are enforced by the compiler inserting specific code to perform the check. E.g., to enforce NO DIVIDE BY ZERO, the compiler might do:

```
if ( denom == 0 ) { get_angry(); }
else { result = num / denom; }
```

```
for i in range( 3 ) :
  print( i )
```

The processing of "range(3)" generates 3 "normal" values 0, 1, 2 and then throws an exception NO MORE VALUES.

The "for" statement in Python catches the NO MORE VALUES exception as the "normal" end of the loop.

```
def runOutOfMemory() :
    a = 2
    while True :
      print( a )
      a = a * a
```

See how far you get before you run out of memory (or get bored).

function --> header body
body --> statementList returnStatement
statementList --> statementList statement
statementList --> <empty>

==========

function --> header body
body --> statementListWithReturn

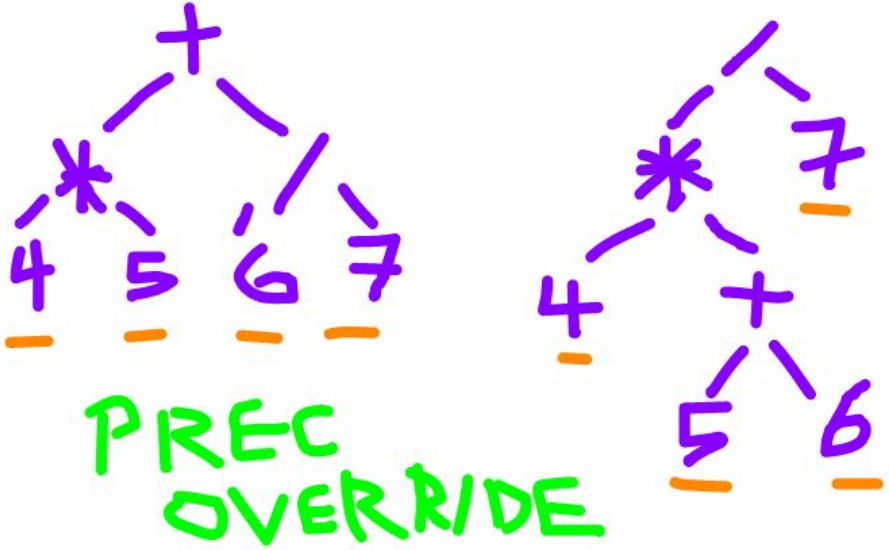
statementListWithReturn -->
    statementListWithReturn  notReturnStmt

statementListWithReturn -->
    statementList  returnStmt

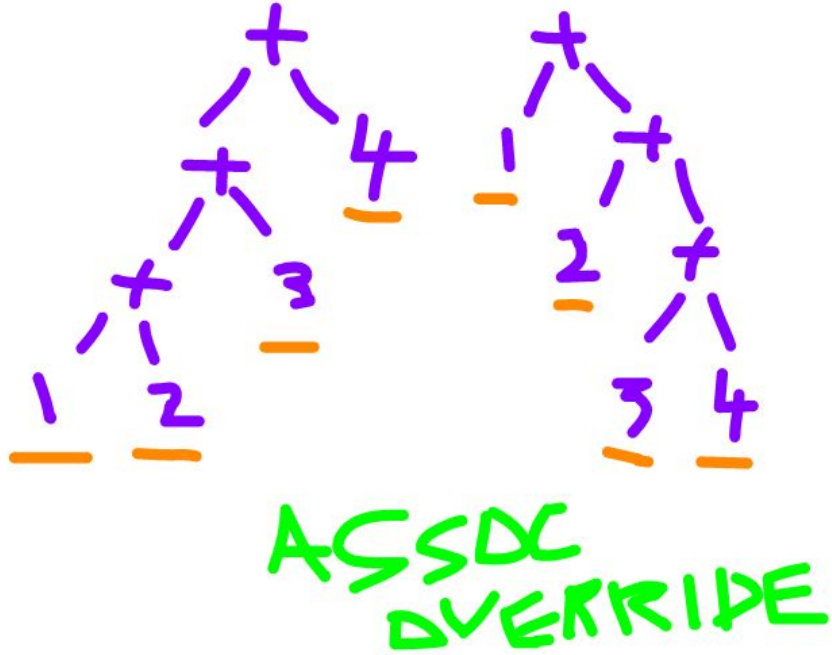
# Semantic Checking of Expressions

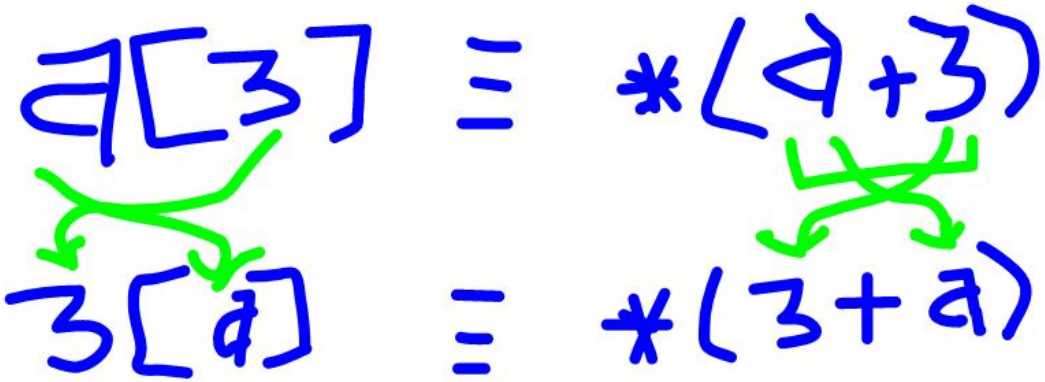
values and operators:

4 * 5 + 6 / 7 vs 4 * ( 5 + 6 ) / 7




1 + 2 + 3 + 4 vs 1 + ( 2 + ( 3 + 4 ))






In C, indexing is written as a[b]. This is DEFINED to mean *(a + b), or dereference the sum of a plus b.

This works because in C, "arrays" are really just pointers (to a spot in memory). Adding to a pointer just moves it the appropriate number of entries.

This is why *a means the same as a[0].

Since addition is commutative, it does not matter which order a and b are written. a[b] is IDENTICAL in meaning to b[a].

Assume: int *a;
a = malloc(10*sizeof(int));
*(a + 3 ) = 5; is the same as
a[3] = 5;
a++; // means move a to the next int,
// so it really gets 4 added to it.

But what about this definition:

```
int a[] = { 1, 2, 3, 4 };
```

What is the type of a?

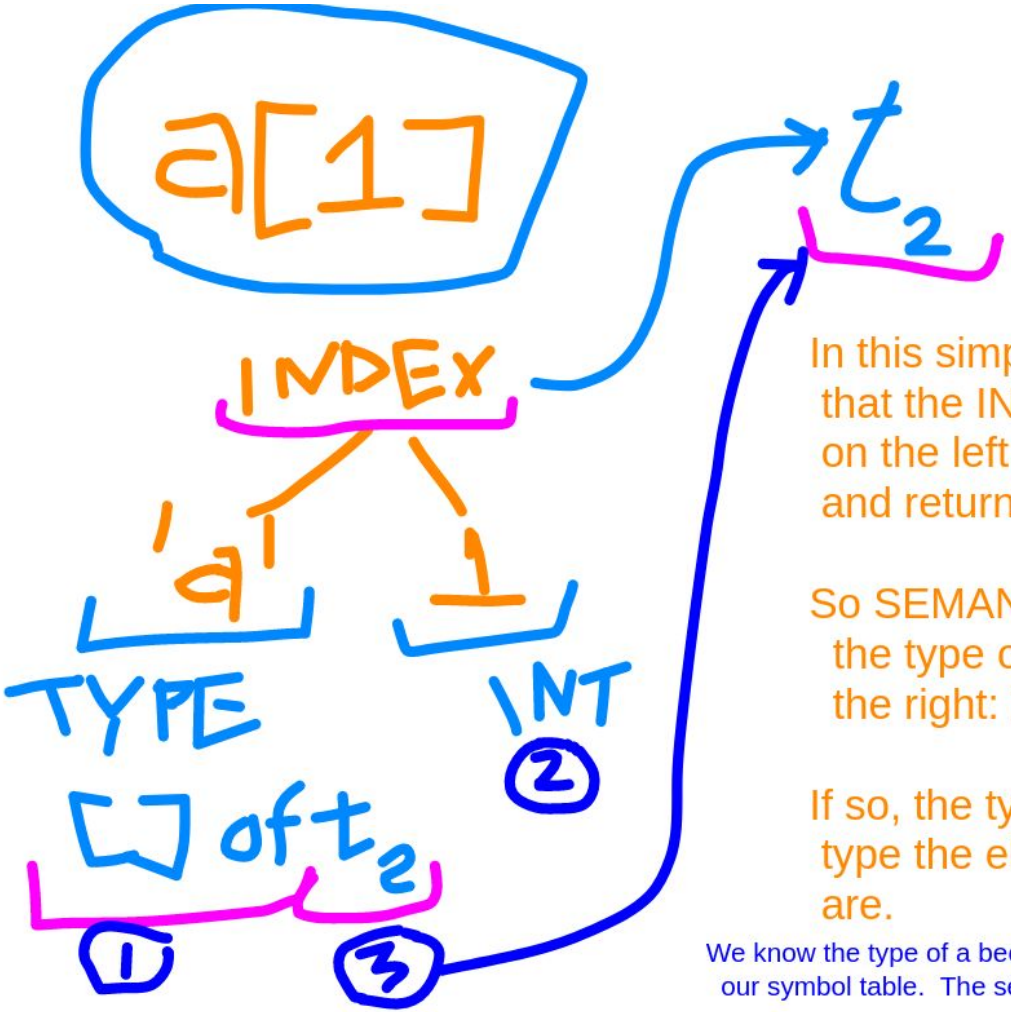
Yep, (int *).

So, a[0] -> 1, a[1] -> 2, etc.

What about a[1231]?

Well, it's whatever happens to be at that spot in memory. Or maybe it's a segmentation error because that memory doesn't exist (or you don't have the right to read it).

What about a[-123]? Yep, same answer as for a[1231]. Whatever happens, happens.

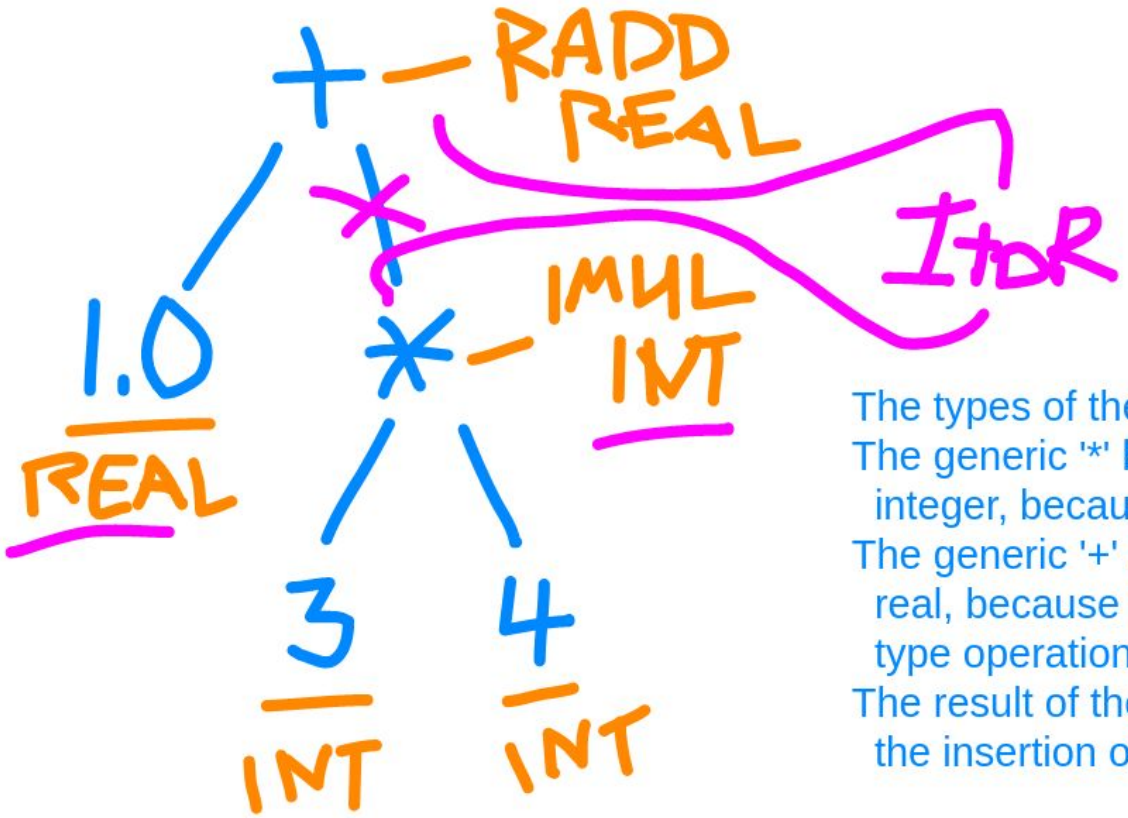



In this simple abstract case, we have said that the INDEX operator has a 1D array on the left and an expression on the right and returns the corresponding element.

So SEMANTICALLY we check the left: is the type of the left a 1D array? We check the right: is the type of the right an int?

If so, the type of the result will be whatever type the elements of the array on the left are.

We know the type of a because it was declared earlier and we saved the info in our symbol table. The semantic analyzer had to query the SYMTAB to get it,




The types of the literals (and the values) are obvious.
The generic '*' becomes INTMUL, with result type integer, because both of its operands are ints.
The generic '+' becomes REALADD, with result type real, because ints get COERCED to reals in mixed type operations.
The result of the INTMUL is converted to REAL by the insertion of an implicit INTtoREAL operator.

## Debugging with flex

Sometimes it's not clear how flex processing is going -- that is, the handling of each character and the rule that is used for the match.

SO, it's possible to tell flex to DEBUG -- use the -d switch when you call flex. This will cause flex to output an enormous number of messages about what it's doing.

---

COMMENTS in source are (almost) always discarded by the lexer. Some persons in 2b created a tok_COMMENT and passed it back.

Why is this a bad idea?

```
// Here's a function.
int  // The return type
funcName  // The name of the function
(  // Get ready for the parameter list
int // The type of the first arg
arg1 )
// and now the body
{  // and so forth ...

int funcName( int arg1 ) { // and so forth
```

---

```
funcdef --> typeName id argList body
argList --> '(' arguments ')'
body -->  '{' stmtsAndDecls '}'
```

```
// OPTIMIZE:SPEED
int func( int arg )
{
   ... a body ...
}
```

There have been compilers that use comments in CERTAIN SPECIFIC PLACES to put DIRECTIVES that control the compilation process.

For example, above, that might be used to tell the compiler to concentrate on SPEED when optimizing that routine.

The compiler gets information that another compiler might ignore -- after all, it's just a comment.

This aspect of a compiler "scanning" the comments is kind of out-of-date. Today, the ATTRIBUTE concept is used instead.

This is now part of the language standard for most language.

The problem was every compiler made up its own way to get directives and they were non-portable in the extreme.

So constructs were added BUT the exact interpretation is left up to the particular compiler.

In C, GCC uses __attribute__
There's also #pragma for doing these kinds of things.

* In a RE in flex, what does . match?
  Anything BUT a newline.

* In a RE in flex, what does [^\n] match?
  Anything BUT a newline.

* In a RE in flex, what does [a|b] match?
  The a character, the b character OR the | character.

* In a RE in flex, what does [-*/\]|[fred] match?
  One of -, *, /, ], |, [, f, r, e, OR d.

* The - and ] and \ characters are SPECIAL in a character class.  - is NOT special if it's the first or last character.  ] is not special if it is escaped with \.  \ is not special if it is escaped with \.  So, to get ] as a character write \].  \\ gets \.  \- gets - or put - first/last.

* 'c' but c NOT ', \n, \.  So '" is WRONG.  '\' is WRONG.  '\'' to get ' as char and '\\' to \ as char.

* ['][^'\n\]['] leading ' then one char that is NOT ', \n, ], [, or '.  What you meant was ['][^'\n\\]['].

* ['].['][']\\[abfgnrtv]['] -- Wrong b/c '\' is accepted.  So is '".

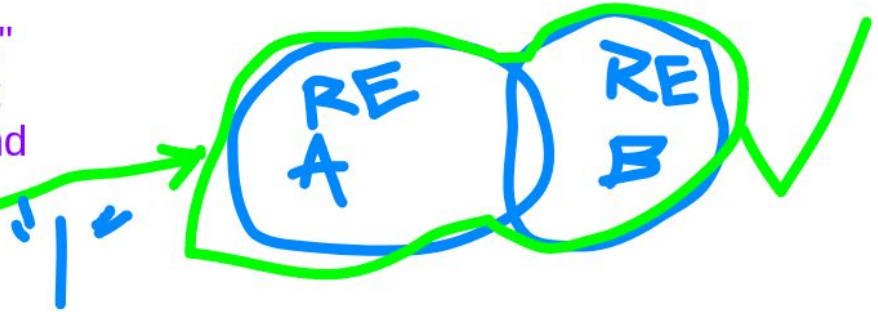
* flex does not care about (nor does it complain about) subsuming matches.

* The RE (.|a|b) has "subsuming" parts; the |a|b is redundant because . will match them anyway.

There were many examples of this in the 2b flex files:  a RE of the form A|B|C|... where the A part matched so much that the subsequent parts were redundant.

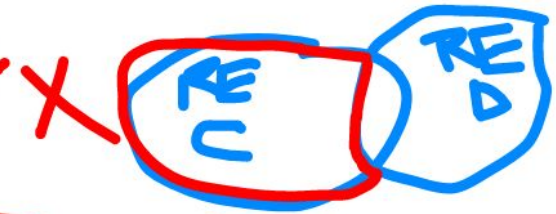
flex doesn't have the concept of "cutting out" except the [^] in character classes. It's not possible in flex to say "match _this_ RE and then _exclude_ what matches _that_ RE".

Getting UNION of two REs is OK.




There's no SUBTRACTION of two REs.

Nor is there the concept of the INTERSECTION of two REs.




It IS possible to write something like:

[A-Z]{-}[AEIOU]


but this is NOT the subtraction of two REs; it is a shorthand for writing a character class. It's like the - shorthand for indicating a range of characters [A-Z] instead of [ABCDEF ... ]

You can't put arbitrary REs with {-} inbetween.

(IN FLEX)

A question that has come back to me is whether "Subtraction" and "Intersection" of two arbitrary Regular Expressions are possible at all (given that there's no "easy" syntax in `flex` to express them). That is, are $L_a - L_b$ and $L_a \cap L_b$ regular languages?

YES! (Remember your Foundations class. :)

Consider the "Complement" of a RE's language: the set of strings over the same alphabet that are NOT in the RE's language.

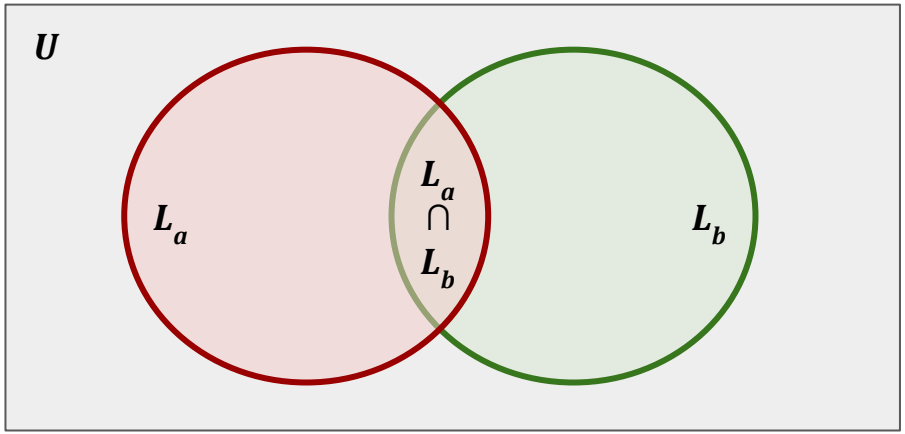
This is regular since we can take the first RE's finite automaton and change all of the accept states to reject states and vice-versa. Presto! We now have an FA that accepts the complement of the original language, thus it's regular. (Remember, we can convert REs to FAs and FAs to REs mechanically.)

Using DeMorgan's law, we have $L_a \cap L_b = \neg( \neg L_a \cup \neg L_b )$, so since "Union" results in a regular language and "¬" ( = "Complement") results in a regular language, the "Intersection" of two regular languages is a regular language.

How about $L_a - L_b$?

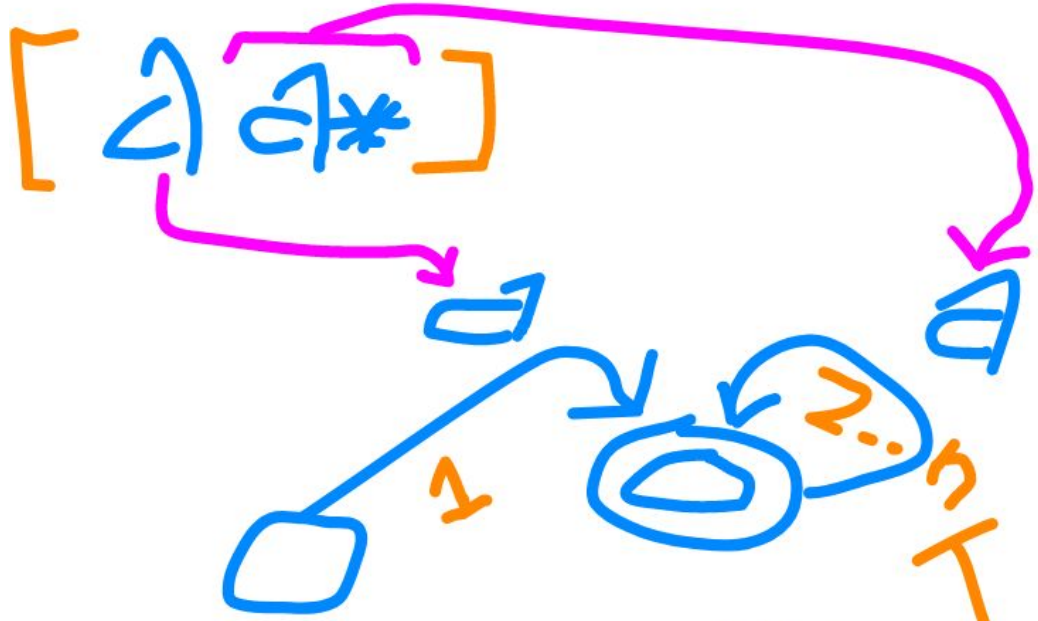
Well, once you understand how ¬L is regular if L is and $L_a \cap L_b$ is regular if $L_a$ and $L_b$ are, showing that $L_a - L_b$ is regular is easy.

Take a look at the following picture. Is the proof method clear to you now? Think about it for a moment.




**Sigh**

OK, what you should have seen in the diagram is that $L_a - L_b \equiv L_a \cap \neg L_b$. Therefore, since both "∩" and "¬" result in regular languages, so will "-".




Regular Expressions, Finite Automata, and Infinity.

A RE itself cannot be infinite in length.
A RE cannot RECOGNIZE an infinite string. (How would you know when it was done?)
A RE CAN accept an infinite number of strings.

A FINITE Automata cannot have an infinite number of states. (It's finite.)

ACCEPT

1. ALL CTR CONSUMED
2. BE IN AN ACCEPT STATE

$\forall n \in \mathbb{N}$

# SHIFT/REDUCE vs REDUCE/REDUCE

SHIFT / REDUCE and REDUCE / REDUCE conflicts are discovered when a parser generator analyzes a given CFG.

They are NOT encountered when a compiler is trying to compile a user's program.

Confusion about this possibilty came up repeatedly in some persons' answers.

These conflicts indicate a difficulty with the grammar itself. A resolution has to be made when the CFG is analyzed or a parser can not be generated. Parser generators tend to
(1) Prefer SHIFT to REDUCE, and
(2) Arbitrarily pick the rule to REDUCE by
(for a REDUCE/REDUCE conflict).

What are two DIFFERENT ways that operator ASSOCIATIVITY can be enforced?

(1) Use the directives supplied by bison / yacc / whatever: %left, %right, and %nonassoc.

(What else gets expressed using these directives? --> PRECEDENCE! The levels of precedence are indicated by the order in which the %left, %right, and %nonassoc directives appear:

```
%left '+'
%left '*'
%right '^'
```

means '+' and '*' are left associative and '^' is right associative. '^' has the highest prec, then '*', and '+' is lowest.)

(2) Use the design of the production rules to enforce a particular associativity and precedence to the operators.

```
expr  ->  term  |  expr '+' term
term  ->  factor | term '*' factor
factor -> root   | root '^' factor
root   -> id | number | '(' expr ')'
```

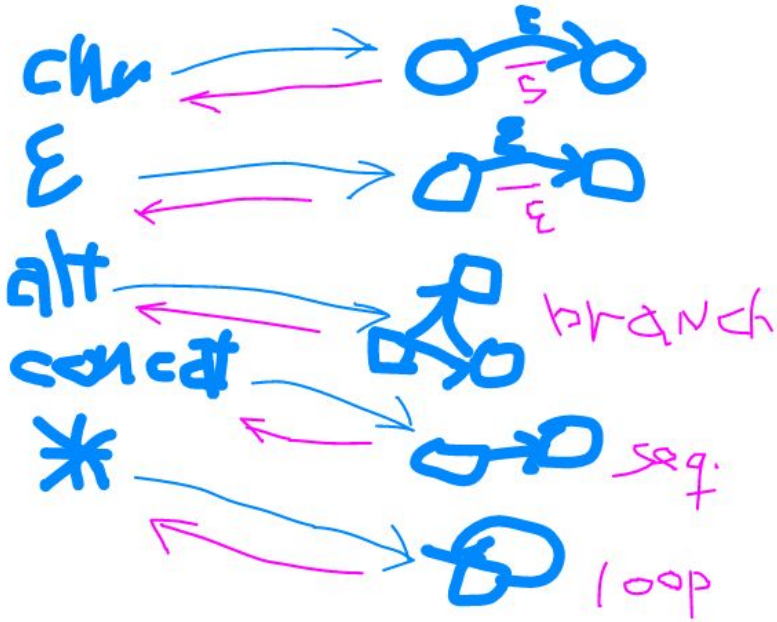
This sets the associativity by which side has the recursion (left or right) and the prec by the level of production rule. (The deeper the rule, the higher the precedence.)

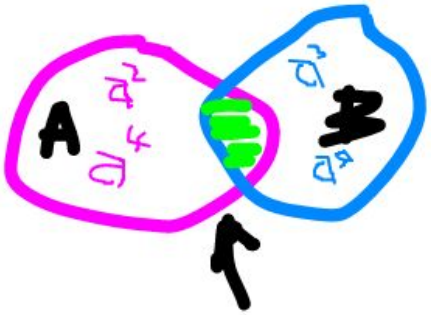
# "REGULAR" language

Regular languages and FA are equivalent.

Anything describable by a Regular Expression is recognizable by a FA and anything recognizable by a FA is able to be written as a RE.

If this is clearly understood, the closure properties are easily derived.






A∧B

A $(aa)^+$ B $(aaa)^+$

A∧B

$a^6$, $a^{12}$, .....

$a^{z^n z^m}$ $n,m \geq 1$

Expression semantic processing is (almost entirely) about the notion of TYPE: names and literals have type and the operations performed have a result type based on the operand type(s) and the particular operator.

In general, expression semantic processing happens recursively, bubbling up from the leaves (which are the names and literals).

Pretty much everything is an operation, even if we don't have a specific user-accessible operator. Some "operators" are "internal". (E.g., the conversion of an integer value to a FP value as the result of coercion.)

We defer the rest of this discussion until we get to more details on TYPE.

type id '(' decl_list ')'
'{' stmt_list '}'

Inside the stmt_list we have access to our own name (we can call ourselves). We also have access to the names in the decl_list.

Both sets of this info are kept in a symbol table --> name, scope, and kind/type.

We then can process semantically each of the statements in the stmt_list.

For simplicity's sake: assignment, while, for, if, return. continue, break

Assignment-> lval '<-' expr

While-> 'while' expr 'do' stmt_list 'end'

For-> 'for' id '<-' expr 'to'|'downto' expr
[ 'step' expr ] 'do' stmt_list 'end'

If-> 'if' expr 'then' stmt_list
[ 'elif' expr 'then' stmt_list ]*
[ 'else' stmt_list ] 'end'

Return-> 'return' [ expr ]

Continue-> 'continue'
Break-> 'break'

# Declarations

as statement-> 'var' id ':' type [ '<-' expr ]

decl in decl_list->  id ':'  type  separated by ','

Assignment-> lval '<-' expr

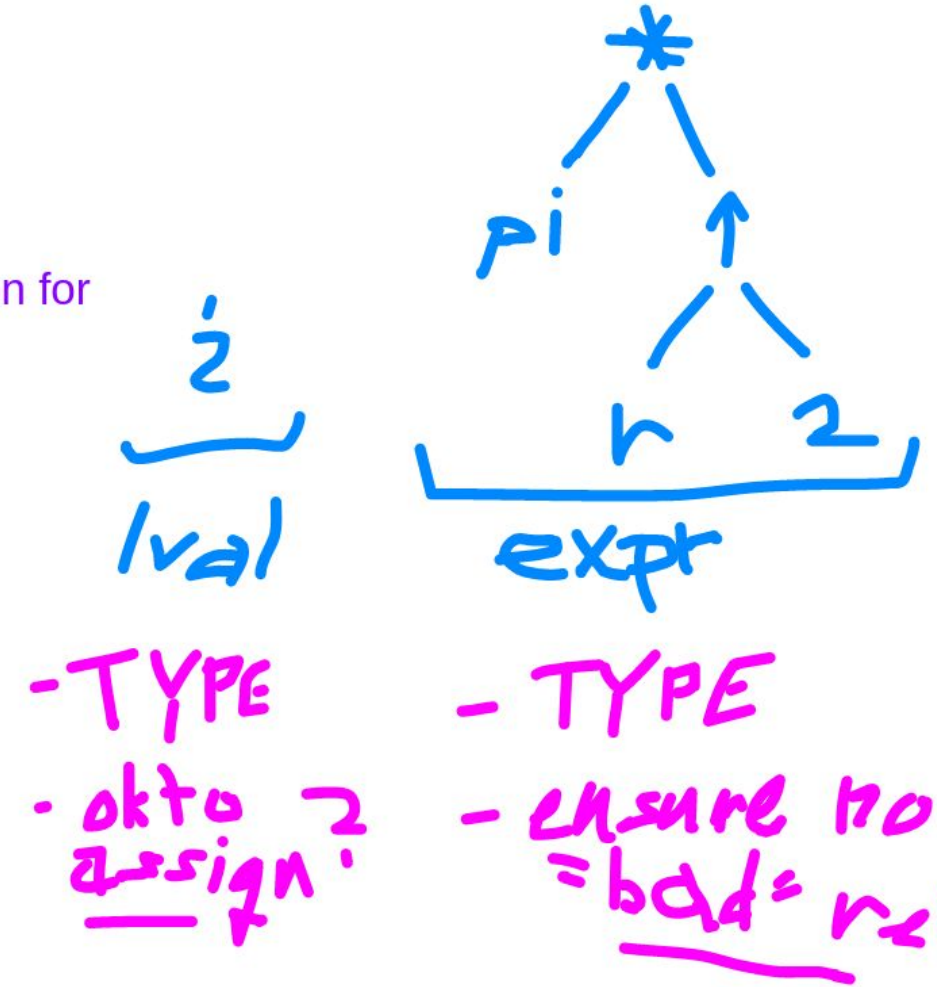
For example,
i <- pi*r^2

What kind of semantic processing goes on for this statement?

* "Semantically" process the expr
* "Semantically" process the lval
* Decide if assignment is legal.

"Legal" -> types are "compatible" and lval is assignable.

(Anything else important for the assignment statement?)



# Expressions: Operators

- Logical
  - OR_LOGICAL, AND_LOGICAL, NOT_LOGICAL
  - EQUAL, NOT_EQUAL
  - LESS, LESS_EQUAL, GREATER, GREATER_EQUAL

returns 0,1

BINARY
UNARY

OR, AND, and NOT take any exprs as operands and think of them as "booleans".
For ints and FP, 0 (F) and != 0 (T).
For string, "" (F), "..." (T).

BO

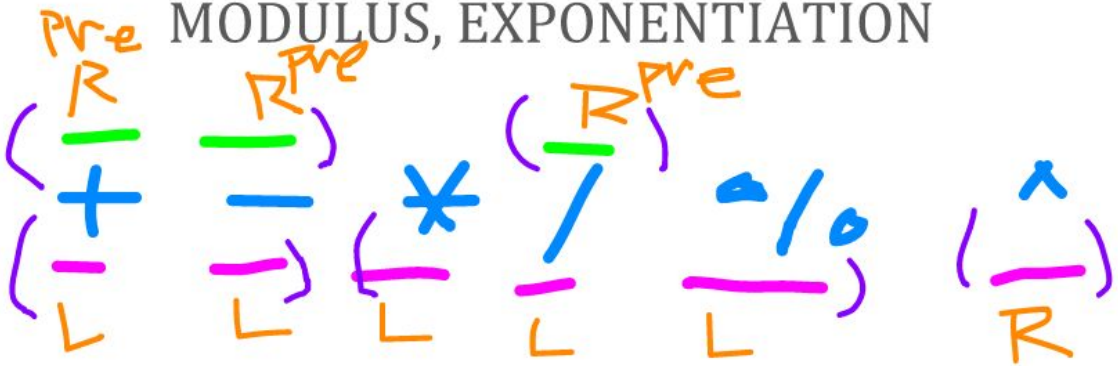
EQ and NE, obvious for ints/FP/string.
LT, LE, GT, GE obvious for ints/FP.
LT, LE, GT, GE are collating sequence for string. Case matters!

More detail req'd for mixing operand types.

# Expressions: Operators

BINARY
UNARY

- Numeric
  - PLUS, UPLUS, MINUS, UMINUS, MULTIPLY, DIVIDE, MODULUS, EXPONENTIATION



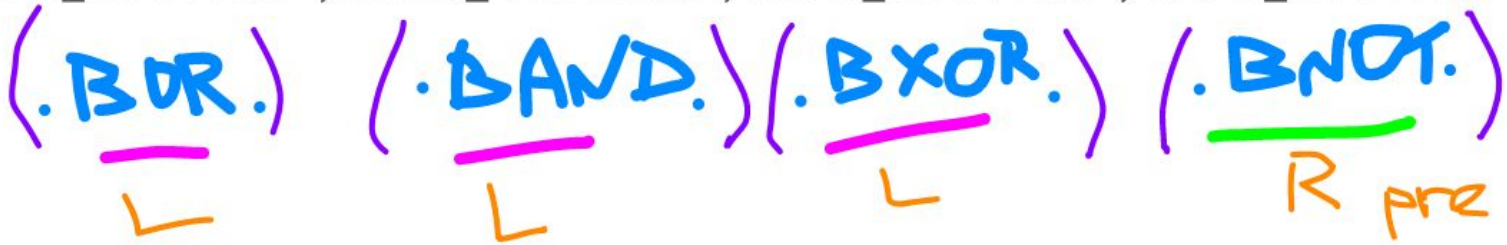

These numeric operators have the "normal" interpretations. If both operands are integers, the result is integer. If either (or both) operands are FP, the result is FP. Unary / is reciprocal: /2.0 -> 0.5. For unary operators, the result is the same as the operand, EXCEPT for unary / which is always FP.

Unary divide /a is DEFINED to be (1.0/a). It's the FP reciprocal of its argument.

# Expressions: Operators

BINARY
UNARY

- "Integer"
  - OR_BITWISE, AND_BITWISE, XOR_BITWISE, NOT_BITWISE




Called the "integer" operators because all operands must be integers and their results are all integers. No FP / string operands allowed.

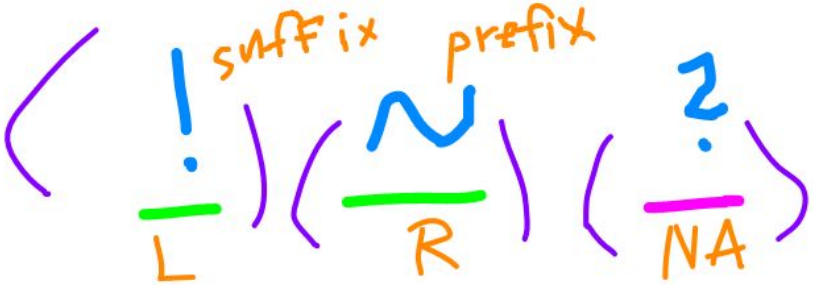
Each operator does the bit-by-bit operation on its operands.

# Expressions: Operators

When the operands to ? are evaled, if the left is GT the right, the two are swapped before computing the R.N. So, 5 ? 1 means the same as 1 ? 5, a R.N. between 1 and 5, inclusive, uniformly distributed.

BINARY
UNARY

- Fun Numeric
  - FACTORIAL, SQUARE_ROOT, RANDOM




Factorial of an integer is an integer. Factorial of an FP number is the gamma function of that FP + 1.0. For example, 1.5! is gamma( 2.5 ).

Square root of an integer is an integer. Square root of an FP is an FP. ~25 evaluates to 5. ~10 evaluates to 3.

Random operator ? takes two operands
If both are integer, returns a uniform random integer between the two operands. E.g., 10 ? 100 returns a random integer from 10 to 100, inclusive, (uniform distribution).

If either operand is FP, it returns a uniformly distributed FP values between the two operands, inclusive. E.g., 1.0 ? 100.0 returns a uniformly distributed random FP between 1.0 and 100.0, inclusive.

E.g., 1 ? 5.0 promotes 1 to 1.0 and then computes 1.0 ? 5.0.

If the operands are in the "wrong" order, ? silently swaps them before tossing the random number.

C operator precedence and associativity info.

Our operator's characteristics might not always agree with this table, but it's a useful reference nevertheless.

| Precedence | Operator | Description | Associativity |
|---|---|---|---|
| 1 | ++ -- | Suffix/postfix increment and decrement | Left-to-right |
| | () | Function call | |
| | [] | Array subscripting | |
| | . | Structure and union member access | |
| | -> | Structure and union member access through pointer | |
| | (*type*){*list*} | Compound literal(C99) | |
| 2 | ++ -- | Prefix increment and decrement[note 1] | Right-to-left |
| | + - | Unary plus and minus | |
| | ! ~ | Logical NOT and bitwise NOT | |
| | (*type*) | Cast | |
| | * | Indirection (dereference) | |
| | & | Address-of | |
| | sizeof | Size-of[note 2] | |
| | _Alignof | Alignment requirement(C11) | |
| 3 | * / % | Multiplication, division, and remainder | Left-to-right |
| 4 | + - | Addition and subtraction | |
| 5 | << >> | Bitwise left shift and right shift | |
| 6 | < <= | For relational operators < and ≤ respectively | |
| | > >= | For relational operators > and ≥ respectively | |
| 7 | == != | For relational = and ≠ respectively | |
| 8 | & | Bitwise AND | |
| 9 | ^ | Bitwise XOR (exclusive or) | |
| 10 | \| | Bitwise OR (inclusive or) | |
| 11 | && | Logical AND | |
| 12 | \|\| | Logical OR | |
| 13 | ?: | Ternary conditional[note 3] | Right-to-left |
| 14[note 4] | = | Simple assignment | |
| | += -= | Assignment by sum and difference | |
| | *= /= %= | Assignment by product, quotient, and remainder | |
| | <<= >>= | Assignment by bitwise left shift and right shift | |
| | &= ^= \|= | Assignment by bitwise AND, XOR, and OR | |
| 15 | , | Comma | Left-to-right |

https://en.cppreference.com/w/c/language/operator_precedence

The "random" operator ? has these semantics:

a ? b -- if both a AND b are integers, this returns a uniformly distributed integer in the range [ MIN(a,b) .. MAX(a,b) ]. (The "[" and "]" indicate the range is INCLUSIVE of the limits.)

How to implement if all one has is a PRNG that generates an integer in the range [ 0 .. n ] ?

low = MIN(a,b)
high = MAX(a,b)
delta = high - low

return PRNG(delta)+low

For example -5 .. 5, delta 5--5 = 10

So PRNG( 10 ) would be in the range [ 0, 10 ] (which is inclusive).

So result would be -5 + [0, 10] = [-5 .. 5 ].

(PRNG(n) means generate a pseudo-random integer in the range [ 0 .. n ], inclusive. The integers are uniformly distributed.)

What about FP? Well, our PRfpG() always returns a uniformly distributed FP number in the range [ 0 .. 1 ], inclusive.
Suppose a and/or b is FP, we then compute low and high and delta the same way.

return PRfpG()*delta + low

0.0 <= PRfpG() <= 1.0
0.0 <= PRfpG()*delta <= delta
adding low to PRfpG()*delta gives us an FP between low and high, uniformly.

# Assignment Statement

- Syntax

  <lval> '<-' <expr>

Two types t1 and t2 are compatible for conversion of t2 TO t1 if:
(1) t1 and t2 are the SAME type.
(2) if t2 is representable as t1.

We consider int convertable to FP and FP convertable to int -- across an <-.

In expression int is convertable to FP but FP is NOT convertable to int. (FP has larger range than int. int is more precise.)

Semantics
---

The types of <lval> and <expr> must be "compatible" -- that is, <expr> must be able to be converted to a value that can be assigned to <lval>.

(We are using 64-bit ints and binary64 FPs. The significand of binary64 is only 52 bits so a "too-large" int will lose up to 11 bits of precision being converted to FP. This happens only for integers w/absolute value > 2^52.)

Strings don't convert anywhere to anything else.

# For Statement

The step expression is the increment (decrement) for the loop counter. It has to have the proper sign (matching 'to' + --or-- 'downto' -).
E.g.,
10 downto 1 step -2 => 10, 8, 6, 4, 2, <exit loop>
1 to 10 step 3 => 1, 4, 7, 10, <exit loop>

- Syntax

‘for’ <id> ‘<-’ <expr> [ ‘to’ | ‘downto’ ] <expr>
[ ‘step’ <expr> ]
‘do’
<stmt_list>
‘end’ ‘for’

<id> is the loop variable -- it's automatically declared with scope equal to the body (stmt_list). The start, stop, and step expressions must be of INTEGER type (no coercion!).

'to' mean +1, 'downto' means -1, but can be overridden by the optional step expr, which must have the proper sign! Also step can't be zero.

Inside, the break and continue statements may be used. break immediately exits the for, continue starts the next iteration, if any.

# If Statement

- Syntax

  'if' <expr> 'then'

  <stmt_list>

  [ 'elif' <expr> 'then'

  <stmt_list> ]*

  [ 'else'

  <stmt_list> ]

  'end' 'if'

Evaluate each <expr> IN ORDER until one is found that evaluates to TRUE. Execute the stmt_list associated with that <expr>. If no <expr> is TRUE, execute the stmt_list of the else clause.

There can be ZERO or more elifs, but ZERO or one elses.

After the appropriate stmt_list (if any) is executed, control picks up after the if statement.

What constitutes TRUE? For an integer, ZERO is FALSE, any other value is TRUE.

For a real, ZERO.ZERO is FALSE, any other value is TRUE.

For a string, the empty string "" is FALSE, any other value (including "false", "0", "0.0", etc.) is TRUE.

(Character constants are integer values so they follow the same rules as integers regarding TRUE and FALSE.)

Remember, character constants are integers.

Each character constant has the value of the corresponding ASCII character.

In particular, '\0' is equal to the ASCII NUL character, which has the integer value 0.

So '\0' counts as FALSE.

# While Statement

- Syntax

  ‘while’ <expr> ‘do’

  <stmt_list>

  ‘end’ ‘while’

The while works pretty much the same way it does in C.

Evaluate the expression. If TRUE evaluate the body. Then do it again, and again, and again, and ...

break and continue can be used in the body of the while. They work the same way as in the body of a for.

The truth value of the expression follows the same rules as for the if statement.

# Miscellaneous Statement: Break

- Syntax

  'break'

Can be used only in the (nested) body of a loop statement of some kind (for, while, <see below>).

When executed, causes the most-enclosing loop statement to exit. Control picks up after that loop statement.

(Maybe we should also have

'do' <stmt_list> 'while' <expr> 'end' 'do'

and

'repeat' <stmt_list> 'until' <expr> 'end' 'repeat'

statements?)

In a do-while, the loop continues as long as the expr is TRUE. The loop exits when the expr becomes FALSE.

In a repeat-until, the loop continues as long as the expr is FALSE. The loop exits when the expr becomes TRUE.

do-while and repeat-until will execute the stmt_list AT LEAST one time. A while-do might NOT execute its body.

# Miscellaneous Statement: Return

- Syntax

  ‘return’ [ <expr> ]

The return statement causes the immediate termination of the running program. The exit status of the program is whatever the expr evaluates to. If the expr is omitted, the status is 0.

The expr must be an integer expression.

# Declaration “Statement”

- Syntax

‘var’ <id> ‘:’ <type> [ ‘<-’ <expr> ]

Declares the <id> as a variable of the given <type> ( integer, real, string ).

The optional initialization expression will be used as the variable's initial value.

If not given, integers are initialized to 0, reals to 0.0, and strings to "".

The type of the expression must be COMPATIBLE with the given type. (This is the same as for an assignment.)

The scope of the declared variable is the stmt_list that it occurs in. At the end of the stmt_list, the variable is destroyed.

Symbol Table --

It's where we keep all info that we need about a -- wait for it -- symbol. It's whatever we need to know to support the remainder of the compilation process.

Symbol tables can be arbitrarily complex or quite simple, depending on the required context.

A strongly related topic is that of SCOPE, because symbol tables have to keep track of information that might be different depending on the scope of the symbol.

All parts of the information might have to be accessible at the same time (and they can't clash with each other).

E.g., it's perfectly OK to write in C:

```
    int a;
a: // a line with a label.

    a = a + 1;   // clearly the int variable
    goto a;      // clearly the label a
```

The two "a" items are in the same scope, but don't clash because they can be used only in different contexts.

Our ASL is quite simple -- we don't have the issues that a more complex language such as C would have with its symbol table.

# Declaration “Statement”

- Syntax

‘var’ <id> ‘:’ <type> [ ‘<-’ <expr> ]

Declares the <id> as a variable of the given <type> ( integer, real, string ).

The optional initialization expression will be used as the variable's initial value.

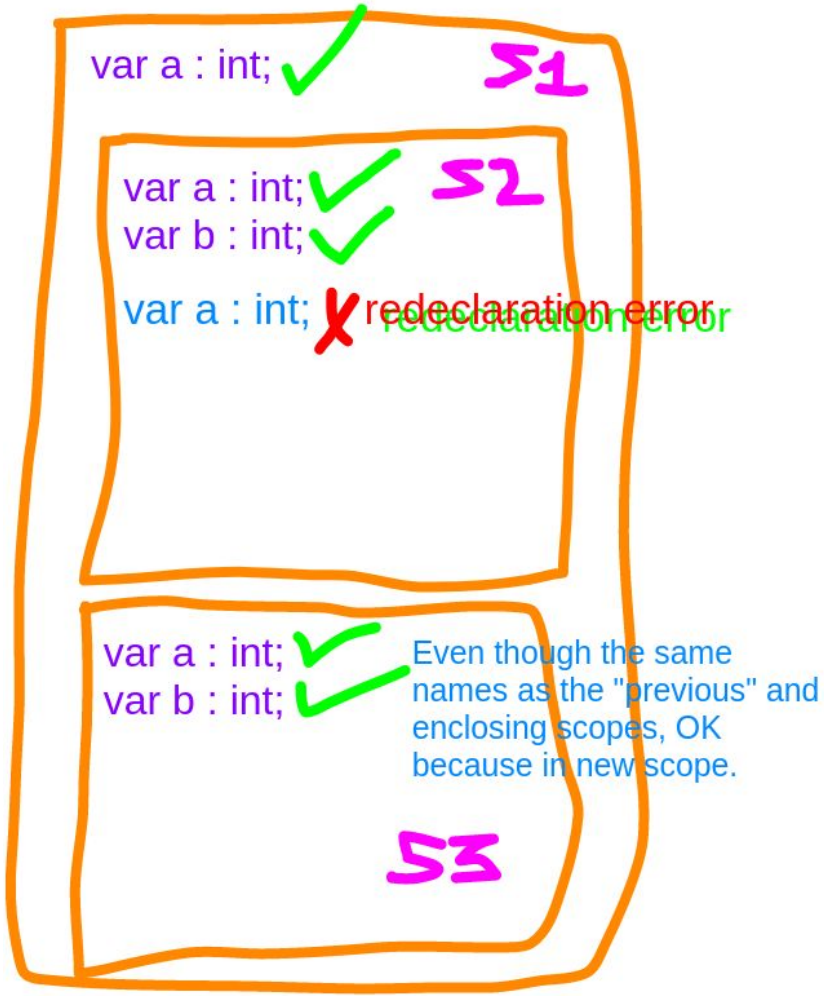
If not given, integers are initialized to 0, reals to 0.0, and strings to "".

The type of the expression must be COMPATIBLE with the given type. (This is the same as for an assignment.)

The scope of the declared variable is the stmt_list that it occurs in. At the end of the stmt_list, the variable is destroyed.

What symtab support required? We need to register that there is now a new binding for the name <id> with the given type.

This binding is in the Current Scope. If there was already a binding IN THE CURRENT SCOPE for this name <id>, a redeclaration error should be reported. (A previous binding being visible is OK as long as its in another (enclosing) scope.)



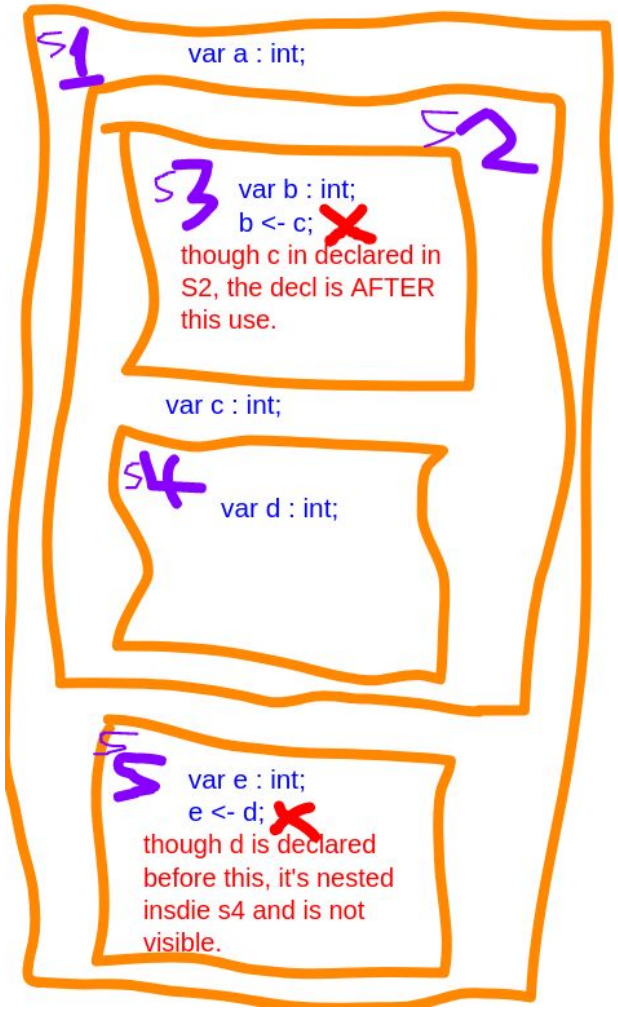

We need to be able to UNIQUELY identify any entry in the symbol table -- just a name is not sufficient. In the example, there are three (legal) "a"s and two "b"s.

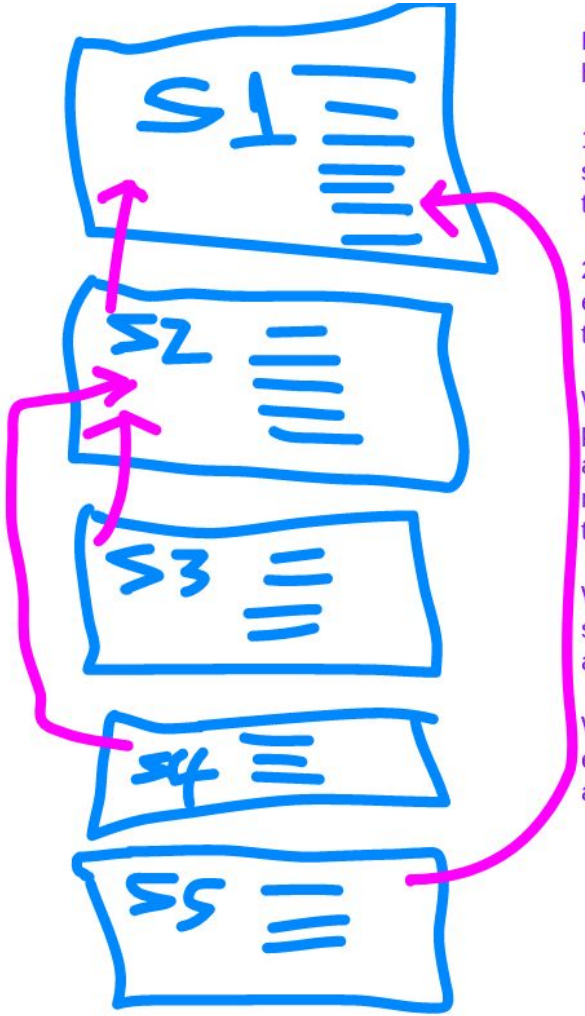
They are distinguishable because they are in different scopes.

For each SCOPE, we need to keep a set of entries, including for each (at least) its name and type.

Each time we "enter" ("create") a new scope, we have to give it a unique name so that we can distinguish bindings made in it from all other bindings.

When the scope is "exited" ("closed"), we have to keep it around for future reference.




For each of the scopes, we have to remember:

1. The parent scope -- the scope that directly encloses this one.

2. The binding information for each name that is "declared" in this scope.

We need the parent scope pointer because in this scope, a reference might be made to a name that was not declared in this scope.

We then search the parent scope for it (and then its parent and so forth).

We get a NOT DECLARED error if no enclosing scope has a declaration for it.

```
Scope:
    int         uniqueID;
    Scope *next;
    Scope *parent;
    Entry   *entries;


Entry:
    int       uniqueID;
    Entry *next;
    char  *name;
    int       type;
```

Scopes and Entries need a unique ID so they can be distinguished more easily from all other scopes and entries. (Some implementors use memory addresses as unique IDs, but ints are better because they can be assigned sequentially and are more easily recognized.)

Scopes are kept in a linear list (using the next ptr) to facilitate logging and searching. (Same thing for the Entry structure.)

Scopes need to know their parents so a hierarchical search for a binding can be made.

Entries have the name and the type. For ASL, type is pretty simple, integer, real, string, so an int suffices.

The last point is just when does a Scope get "created" and when is it "destroyed"?

(1) There is a scope representing the entire contents of a file -- the so-called "file" scope.

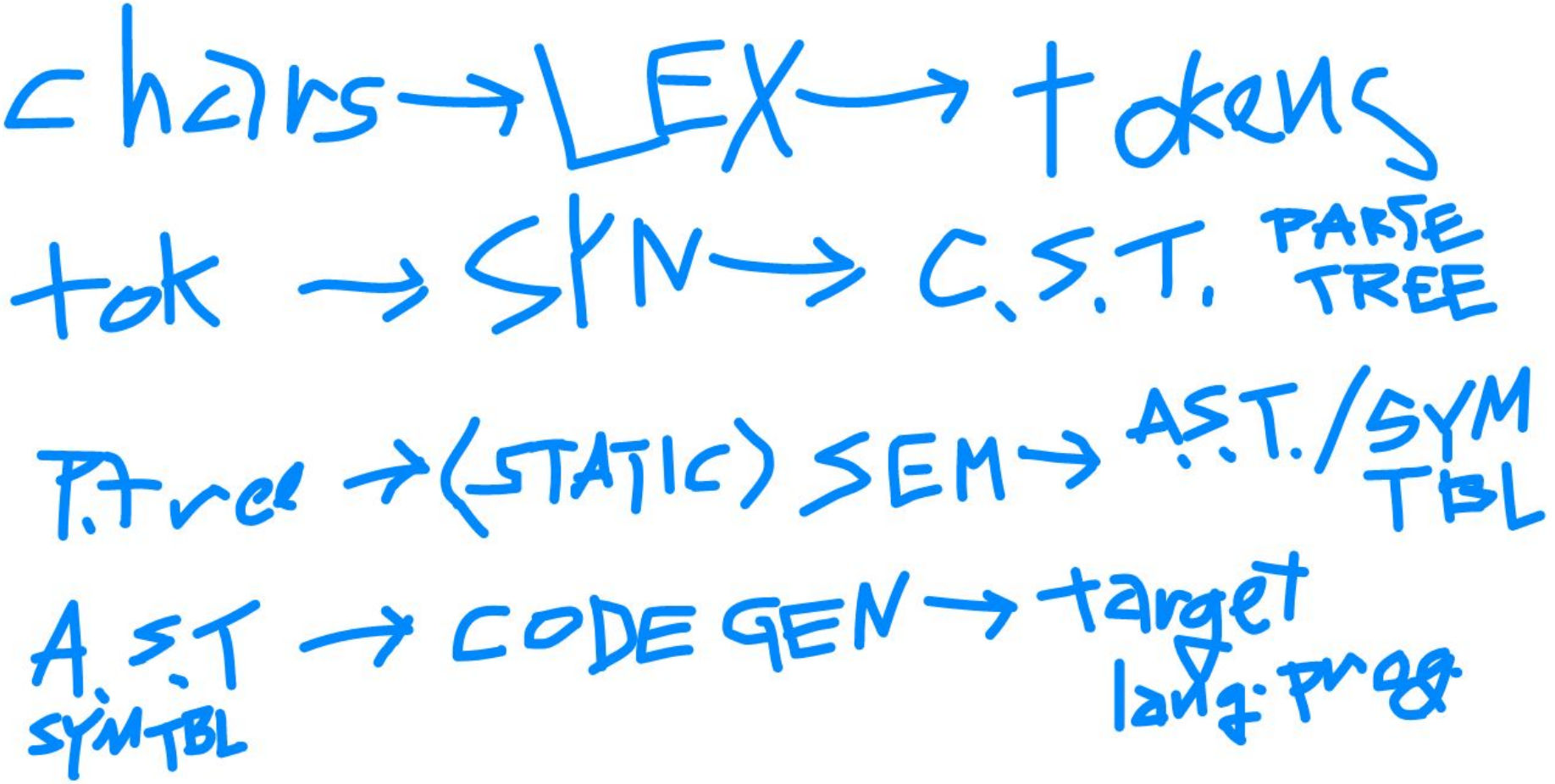
(2) The "body" (stmt_list) of a compound statement is a scope. This includes the if statement (the THEN clause, each of the ELIF clauses, and the ELSE clause).

The body of a for or a while statement. (And the body of a repeat-until.)

A Scope gets created at the very beginning of the file processing and closed at the end of the file.

A scope gets created at the beginning of each "body" in a compond statement and closed at the end of that statement.

Whenever a name is referenced, the current scope is searched and if the name is not found, the parent scope is searched, and so forth until either the name is found or you run out of scopes.


chars → LEX → tokens
tok → SYN → C.S.T. PARSE TREE
P.Tree → (STATIC) SEM → A.S.T./SYM TBL
A.S.T SYMTBL → CODE GEN → target lang: prog

That previous description is True In General, but the real world is a lot messier.

"The difference between Theory and Practice is that in theory there is no difference whereas in practice there is."

The net here is that there's always a bunch of messy details and special cases that need to be taken care of.

We saw this when we spoke about taking a Context Free Grammar and making it into a parser. There are automatic tools (bison, ply, yacc, Antlr, etc., etc.) that will do this, but there are messy details.

In particular, there are restrictions placed on the kind of grammar that can be processed, how efficiently it can be processed, etc., etc.

What does this mean (in C):

```
fred * barney;
```

OK, lots of you want to say that is meaningless.

Suppose it looked like this:

```
typedef int fred;
fred * barney;
```

To correctly process the 2nd line, semantic processing has to be done for the 1st line! (Recognize the <TYPENAME> and feed that back to the lexer.)

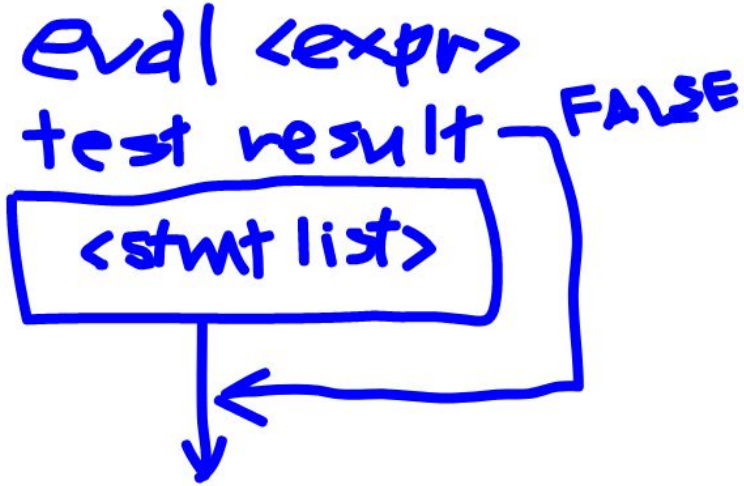
Is it still meaningless?

The difficulty is that "fred" might be an <ID> or it might be a <TYPENAME>. Lexically, these two categories are identical. This is an example of how "messiness" can creep all the way back in to the lexer.

Code generation from a Pattern-based point of view: for a given construct in the source language (as processed by the semantic analysis phase) a pattern in the Target language is generated.

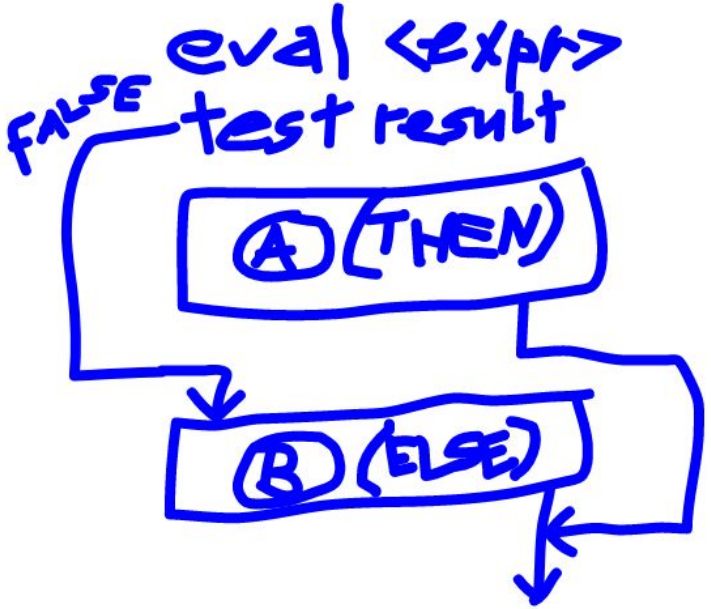
For example, let's consider a simple IF-THEN statement:

IF <expr> THEN <stmtlist> END IF




The block that gets generated has one entry and one exit. We do the same pattern process for every construct that the semantic phase generates. Their composition is the program in the Target language.

A more complex example: IF <expr> THEN <stmtlistA> ELSE <stmtlistB> END IF

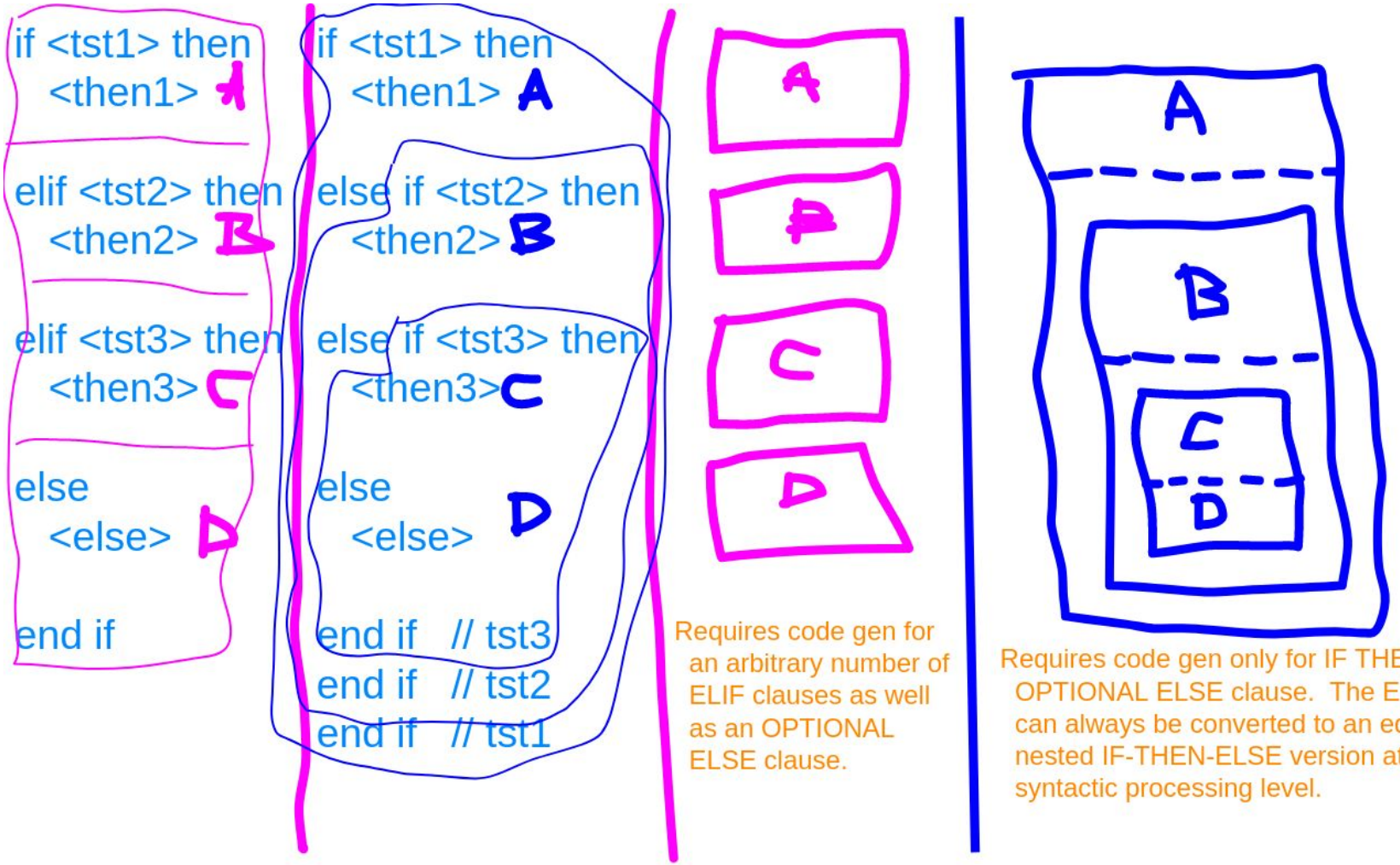


By "composition" we mean that a series of statements can be code generated by sequencing the code generation of each of the statements serially.

So codegen( A, B, C, D ) ends up being codegen( A ), codegen( B ), codegen( C ), codegen( D ).

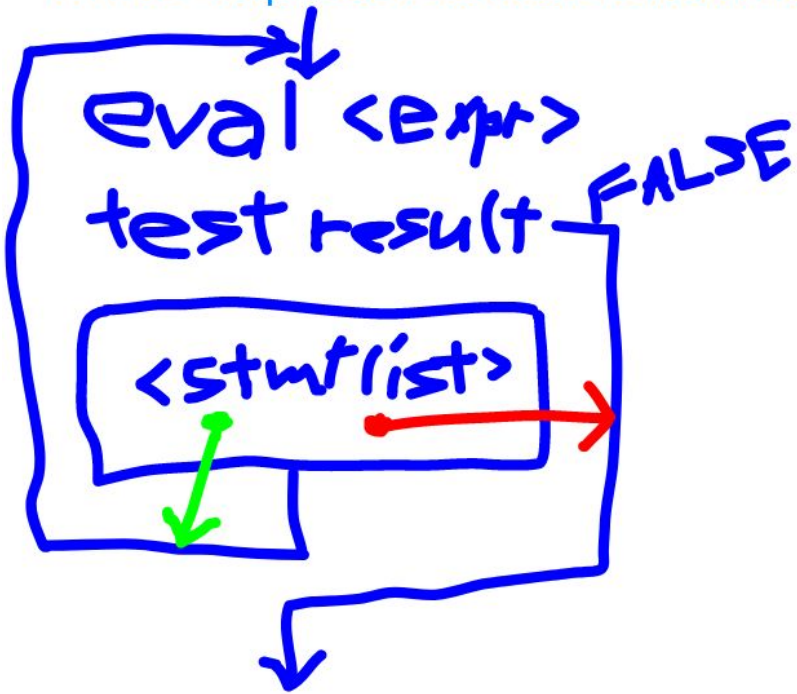
For compound statement ( ifs, loops, anything with an enclosed <stmtlist>), we construct ACCORDING TO THE PATTERN FOR THAT STATEMENT, control flow that uses the code generated for each PART of the compound statement.

In the IF statements examples, we generated code for the test expression. We then tested the result obtained by evaluating that expression and used the result to select which <stmtlist> is evaluated (THEN or ELSE).


if <tst1> then
<then1> A
elif <tst2> then
<then2> B
elif <tst3> then
<then3> C
else
<else> D
end if
if <tst1> then
<then1> A
else if <tst2> then
<then2> B
else if <tst3> then
<then3> C
else
<else> D
end if // tst3
end if // tst2
end if // tst1
A
B
C
D
Requires code gen for an arbitrary number of ELIF clauses as well as an OPTIONAL ELSE clause.
A
B
C
D
Requires code gen only for IF THEN with OPTIONAL ELSE clause. The ELIF style can always be converted to an equivalent nested IF-THEN-ELSE version at the syntactic processing level.

Consider the WHILE-DO statement:

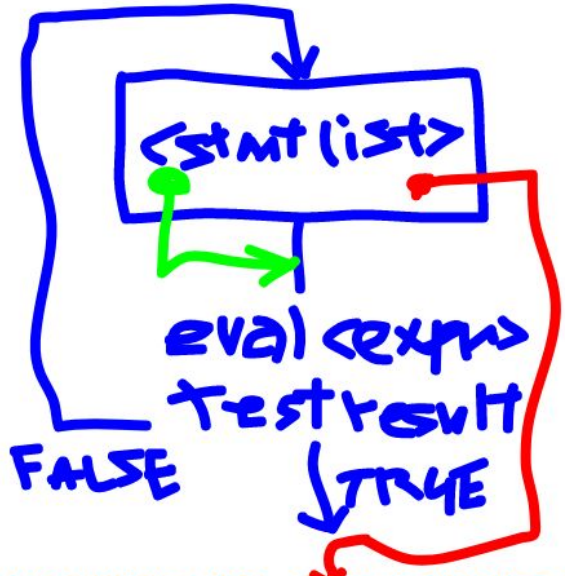
WHILE <expr> DO <stmtlist> END WHILE




Any break statement goes to the same place as the FALSE <expr> eval. Any continue statement goes to the eval <expr> at the beginning.

A complication is that it's not "ANY" break or continue -- it's only the ones that are not inside any nested loops.

That is, breaks and continues affect the MOST enclosing loop (while, for, repeat).




REPEAT <stmtlist> UNTIL <expr> is similar to the while. The test ordering is after instead of before.

The FOR statement is more complex:

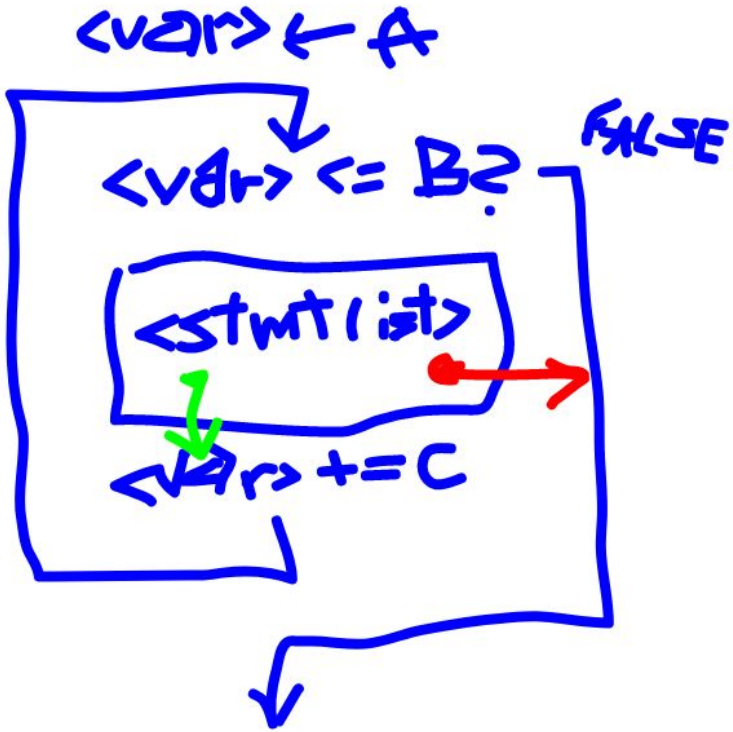
FOR var = <start> TO/DOWNTO <stop>
STEP <step> DO <stmtlist> END FOR


We evaluate <start>, <stop>, and <step> ONCE at the beginning.

<step> defaults to -1 for DOWNTO, 1 for TO.

The test is shown as "<= B". This is the case for TO. If the loop is DOWNTO, the test is ">= B".

It's an error for STEP to be the "wrong" sign or zero.




As with the other loops, continue starts the next iteration of the loop instantly; break exits the loop instantly.

The declaration statement is of two parts:

VAR <id> : <typename> <- <expr>

(1) is "<id> : <typename>" : This is handled at compile time; it's an entry in the symbol table.

(2) is "<- <expr>" : This is handled at runtime so requires code gen. The expression is evaled and the value moved to the memory associated with <id>.

For example,

```
while 1 do
    int i = 7;
    i = i + 1;
    write( i );
end while
```

Every time the decl is evaled, the initializer happens.

What output should this loop generate? There are two possibilities,

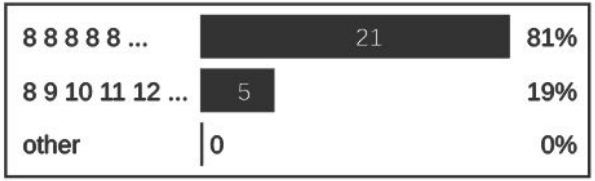
8 8 8 8 8 8 8 ...

or

8 9 10 11 12 13 14 ...

C, C++, Python all generate

8 8 8 8 8 ...



Assignments are straightforward,

<lval> <- <expr>

The <expr> is evaluated and the result (which might have to be coerced) is copied into the memory associated with <lval>.

---

READ and WRITE are special in that they require calls to the runtime system and/or the O/S.

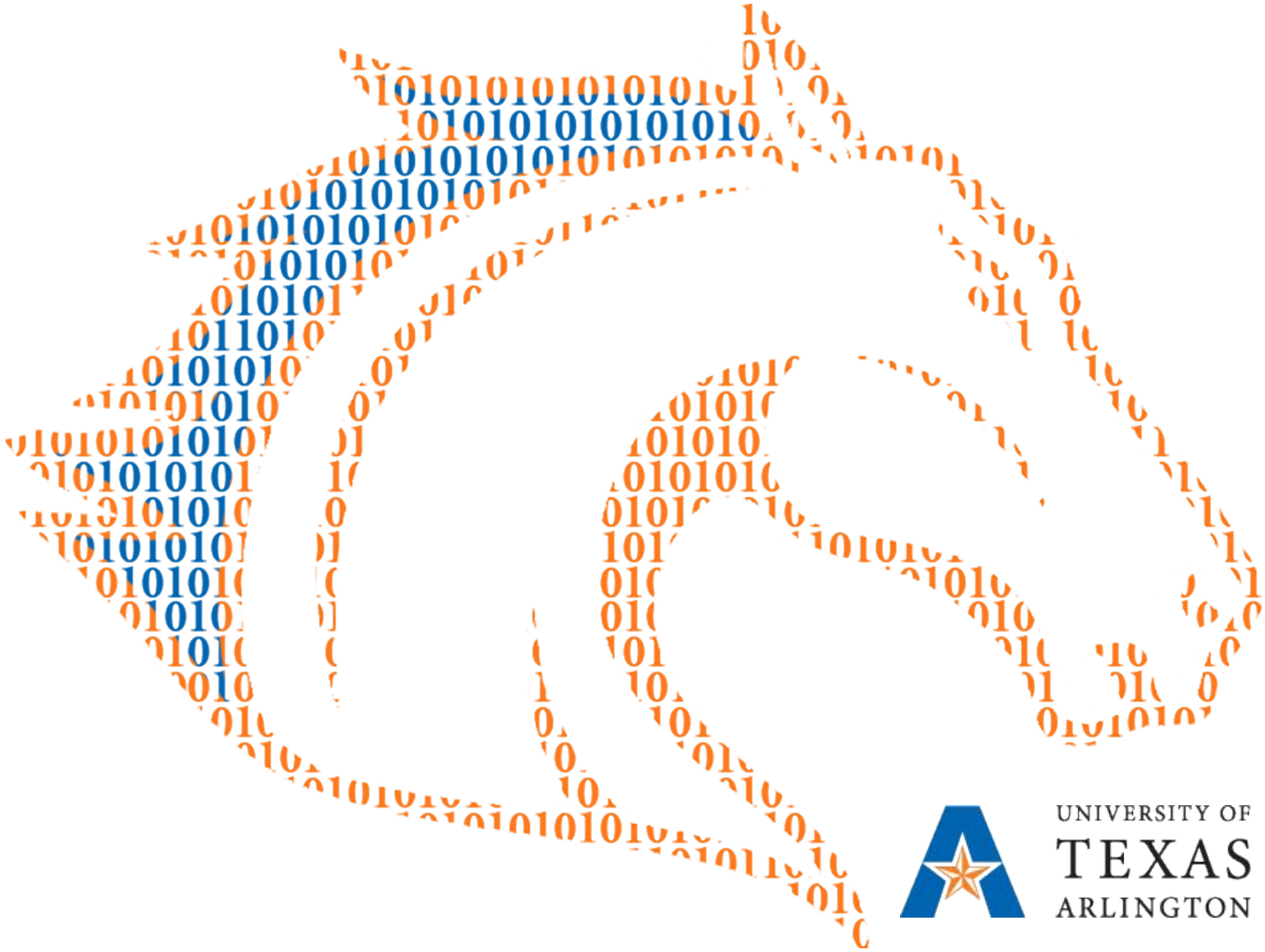
WRITE( <expr1>, <expr2>, ... )

For each <exprn>, evaluate the expression and then convert it (if necessary) to characters and then print the characters.

For READ( <lval1>, <lval2>, ... )

For each <lvaln>, read from the console and convert the characters (if necessary) to the proper form for the type of the <lval>.

It's an error if the characters can't be converted.





UNIVERSITY OF TEXAS ARLINGTON

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING