1. Process of compilation
   A. Chars -> LEX -> Tokens
   B. TOKENS -> SYN -> Concrete Syntax Tree (Parse Tree)
   C. Parse Tree -> (STATIC) SEM -> Abstract Syntax Tree/Symbol Table
   D. Abstract Syntax Tree -> CODE GEN -> Target Language Program

2. Advantages of compilers over interpreters:
   - Code generated by a compiler generally works faster than code generated by an interpreter
   - Once a code is compiled, it can be reused multiple times

3. (a) In the production rule “term : term * factor”, why is “term” on the left of the “*” operator?
   (b) What would change if the rule had been written as “term : factor * term” instead?
   - It acts as left recursion
   - It would now change to right recursion

4. Does always using one kind of derivation (right or left) avoid ambiguity problems in CFG?
   - No

5. Types of operator associativity:
   - Left-to-right, Right-to-left, and Non-associative or does-not-apply

6. LL(1) -- Left-to-right scan of i/p stream, Leftmost derivation, 1 token of lookahead

7. LR(k) errors:
   - Reduce/reduce conflict when the top of the stack matches with RHS of multiple production rules
   - Shift/reduce conflict when it is not clear whether to shift a token onto the stack or reduce whatever is at the top of the stack

8. What is not possible in an RE that is possible in a CFG?
   - RE is limited in expressiveness and has no ability to recurse
   - RE cannot define strings that are required to have nested parentheses, brackets, etc or matching pairs
   - RE uses DFA that cannot be used to recognize nesting that is too deep because there won’t be enough states in the DFA. But this is possible with a

CFG.

9. Disadvantages of hand-coded compilers:
   - Tedious to write
   - DIfficult to extend and scal
   - Error-prone

10. The semantic analysis phase of a compiler converts:
    - parse tree into an intermediate form

11. Precedence and associativity of operators in a programming language are:
    - Syntactic properties

12. (a) Source-to-source compiling with an example language:
    - Using one compiler to generate a relatively high-level intermediate program and using another compiler to get to the final target program
    - C++ was originally implemented this way

13. Terminal vs Non-terminal symbols in CFG?
    - Terminal symbol represents a single element of the language
    - Non-terminal symbol represents a group of terminal symbols defined by production rules

14. What are bison's precedence directives used for?
    - %left to represent left associativity
    - %right to represent right associativity

15. Is a misspelled keyword a lexical error? Why?
    - No, because it can be an identifier in the program. So, it is a syntactic error

16. Listing grammars in a hierarchy (such as the Chomsky) differentiates them in at least four different ways:
    - Parsing complexity, Expressiveness, Resource requirements, Recognizing Automaton
    - In terms of expressivity, the types of grammars are:
        - Regular, CFG, Context-sensitive, and Recursively enumerable

17. Just because an FA is finite, does the language it accepts have to be finite? Why?
    - No, because as long as the input string is of finite length and satisfies the

pattern, it is accepted.
- e.g. S -> aS, S -> (epsilon) is an example of an infinite language that could be accepted by an FA

18. An example when interpretation may still be required for a compiled language:
    - When “on-the-fly” code generation is desired

19. Bison:
    - Bison is a parse generator for an LALR(1) context-free grammar

20. Two ways associativity may be enforced in a CFG?
    - Use the directives supplied by the parse generator (bison or yacc) such as %left, %right, and %nonassoc
    - By enforcing particular associativity defining which side the recursion should be (left or right)

21. Enforcing precedence in a CFG?
    - By having nested rules that permit the repetition of only certain operators at each precedence level. (the deeper the rule, the higher the precedence)
    - e.g. add_op is looser than mul_op as it is written higher in CFG definition

22. What kind of conflict occurs in the following grammar:
    A -> B c d
    A -> E c f
    B -> x y
    E -> x y
    - Reduce/reduce conflict occurs
    - Because when processing “c” from the input stream, the top of the stack matches with the first two rules in the definition.
    - This conflict can be avoided either by refactoring & restating the production rules or by increasing the number of lookahead tokens.

23. For lexical analysis, which is easier to use, DFA, or NFA?
    - DFA is easier to use
    - Although DFA is easier to use, it is not easier to generate a DFA. Hence, first, an NFA is generated that is converted to DFA for use in lexical analysis.

24. Lexical Errors:
    - Badly formed literals, Illegal characters

Syntactic Errors:
- Misspelled keywords, Improper structure, Bad statement and expression construct

Semantic Errors:
- Undeclared identifiers, mismatched function calls, etc

25. Regular languages and FA are equivalent
- Anything describable by an RE is recognizable by an FA and anything recognizable by an FA is able to be written as an RE

26. Operators (in Expressions)
A. Logical: OR, AND, EQ, NE, LT, …. , NOT
B. Numeric: Plus, Minus, Divide, … , Reciprocal, ++, --
C. Integer: OR_BITWISE, AND_BITWISE, XOR_BITWISE, NOT_BITWISE
  - No FP or String operands allowed for these operators
D. Fun Numeric: Factorial, Square_root, Random

27. PRNG(n) == random number in [0, n]

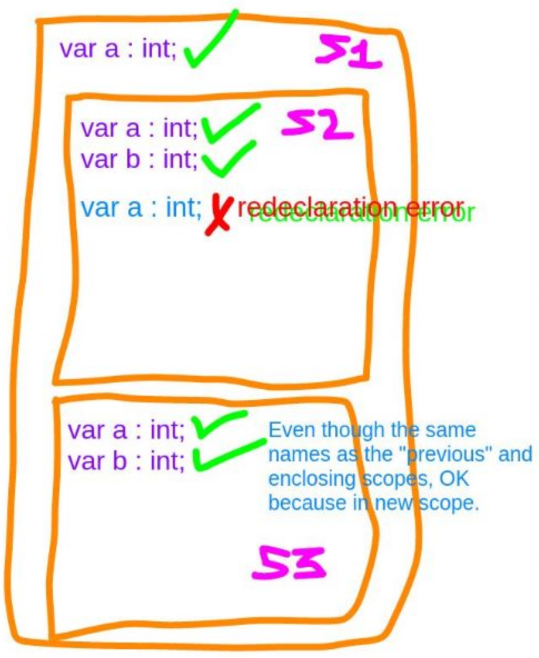
28. Significand of binary64 is only 52 bits
- This means a too large int will lose up to 11 bits of precision being converted to FP (happens when an integer has value > 2^52)
- Strings don't convert anywhere to anything else.

29. Symbol Table
- A table where we keep all info needed about a symbol that might be needed to support the remainder of the compilation process
- It also has 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 eachother)
E.g. It's ok to write like this 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

30. Declaration statement

- ‘var’ <id> ‘:’ <type> [‘<-’ <expr>]
- First, we need to register in the symbol table that there is a new binding for the name <id> with the given type
- The binding is in the current scope. So, 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 it is in another (enclosing) scope

31. SCOPES


S1
var a : int; ✓
S2
var a : int; ✓
var b : int; ✓
var a : int; ✗ redeclaration error
var a : int; ✓
var b : int; ✓
Even though the same names as the "previous" and enclosing scopes, OK because in new scope.
S3

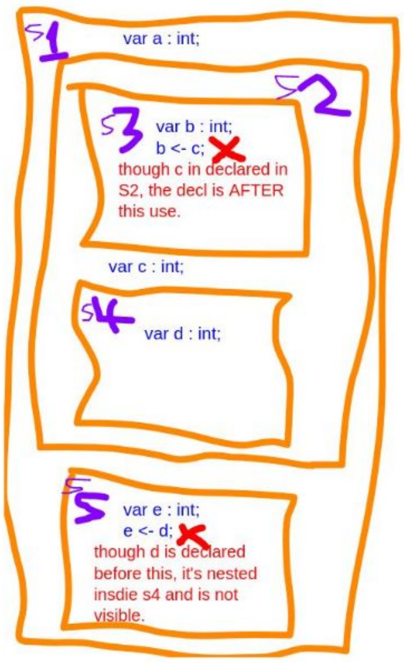
We need to be able to UNIQUELY identify any entry in the symbol table -- just a name is not sufficient. In the example, there are three (legal) "a"s and two "b"s.

They are distinguishable because they are in different scopes.

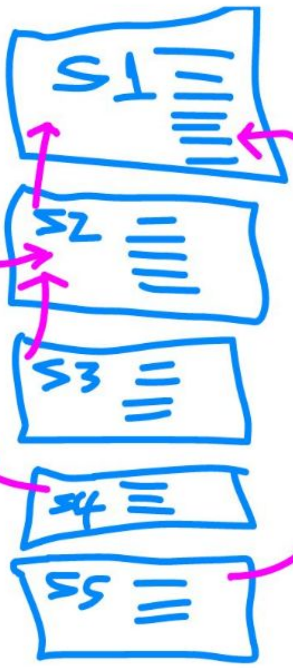
For each SCOPE, we need to keep a set of entries, including for each (at least) its name and type.

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.



S1
var a : int;
S2
S3
var b : int;
b <- c;
though c in declared in S2, the decl is AFTER this use.
var c : int;
S4
var d : int;
S5
var e : int;
e <- d;
though d is declared before this, it's nested insdie s4 and is not visible.



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.

32. Compilation in Real World

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.)

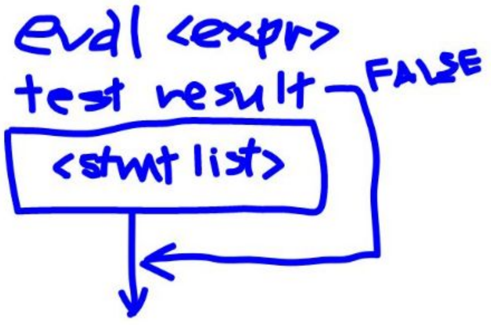
Is it still meaningless?

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.

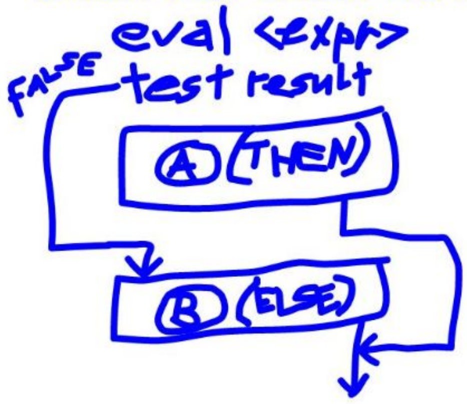
For example, let's consider a simple IF-THEN statement:

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



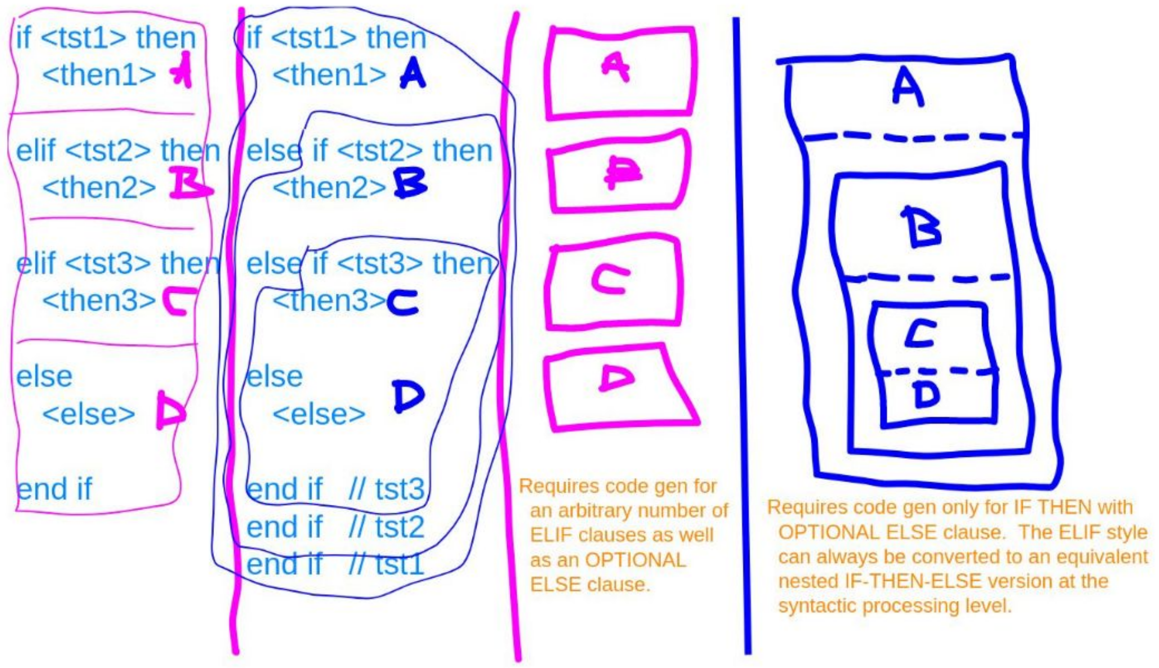
33. Composition in Code Generation

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
```

Requires code gen for an arbitrary number of ELIF clauses as well as an OPTIONAL ELSE clause.

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.

The FOR statement is more complex:

FOR var = $\overset{A}{\text{<start>}}$ TO/DOWNTO $\overset{B}{\text{<stop>}}$
STEP $\underset{C}{\text{<step>}}$ DO <stmtlist> END FOR

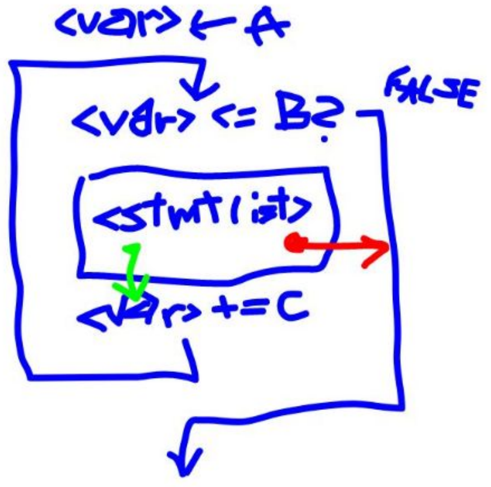
A ← <start>
B ← <stop>
C ← <step>

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>.

C, C++, Python all generate

8 8 8 8 8 ...

For example,

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

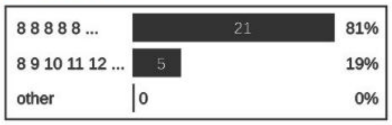
Every time the decl is evaled, the initializer happens.

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 ...
```



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.