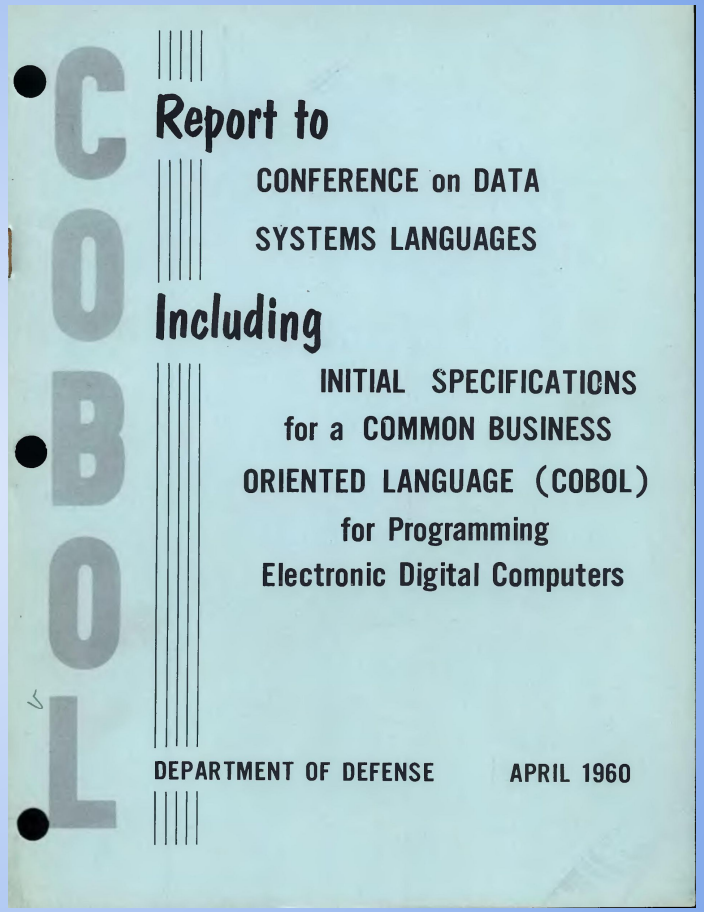
Compilers

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.

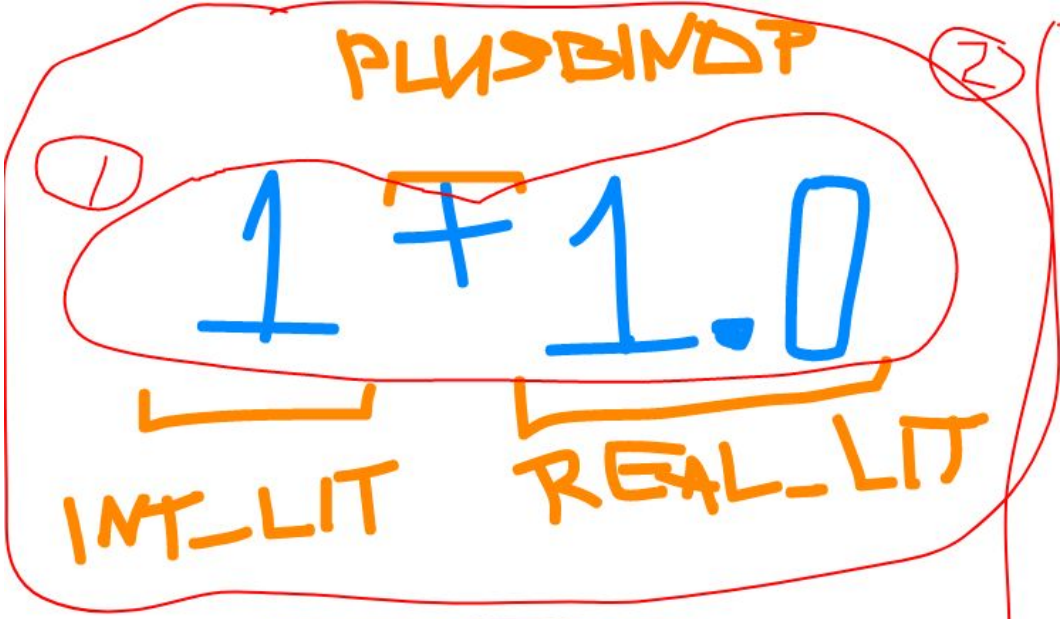
Be mindful of all of this as you review these pages. Tell me if anything is wrong. Thanks!

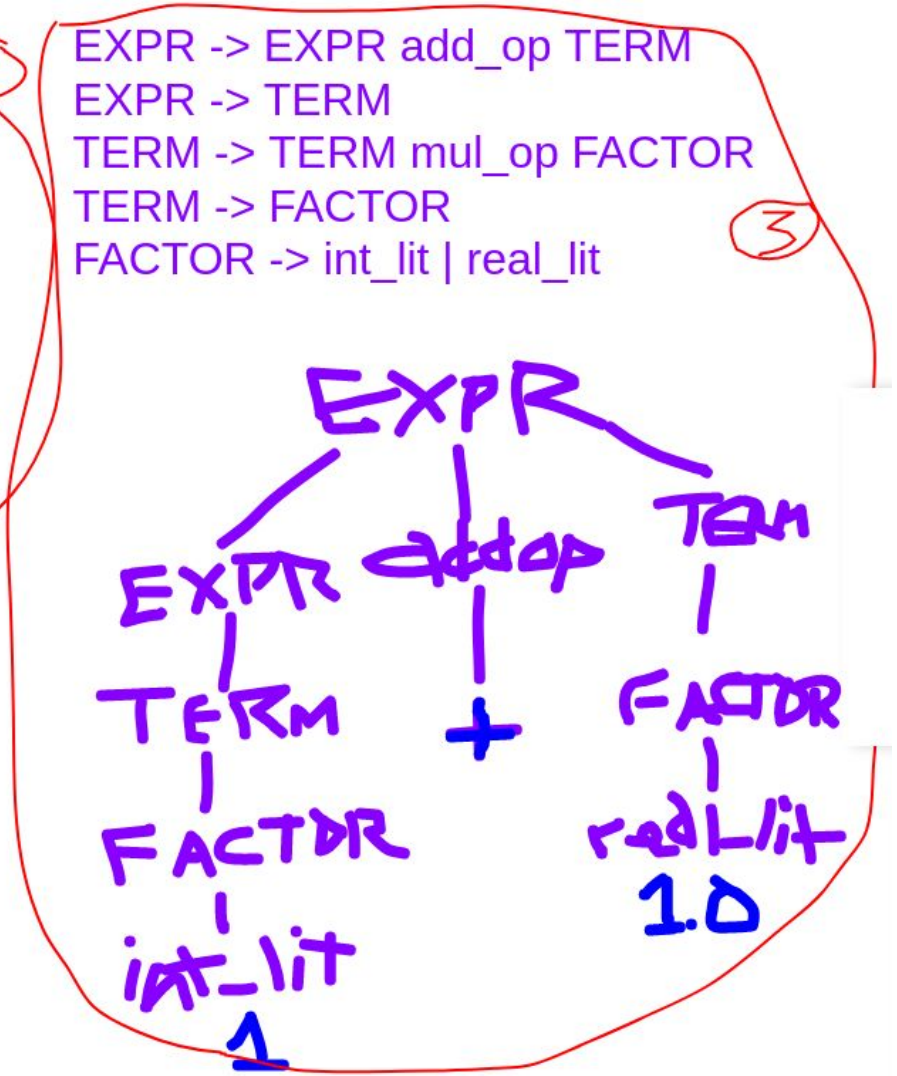
SEMANTIC = PROCESSING

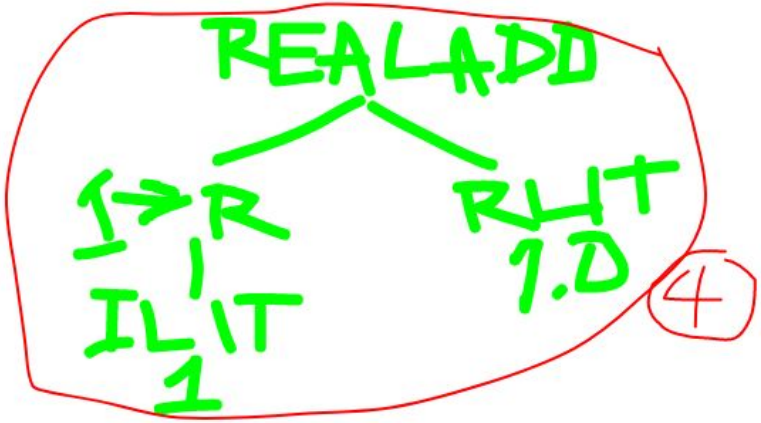
Lexical processing -- we used FLEX to transform Regular Expressions to a scanner.
Regular Expressions are a formal notation that FLEX can process in a mechanical way. Characters became tokens.

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.

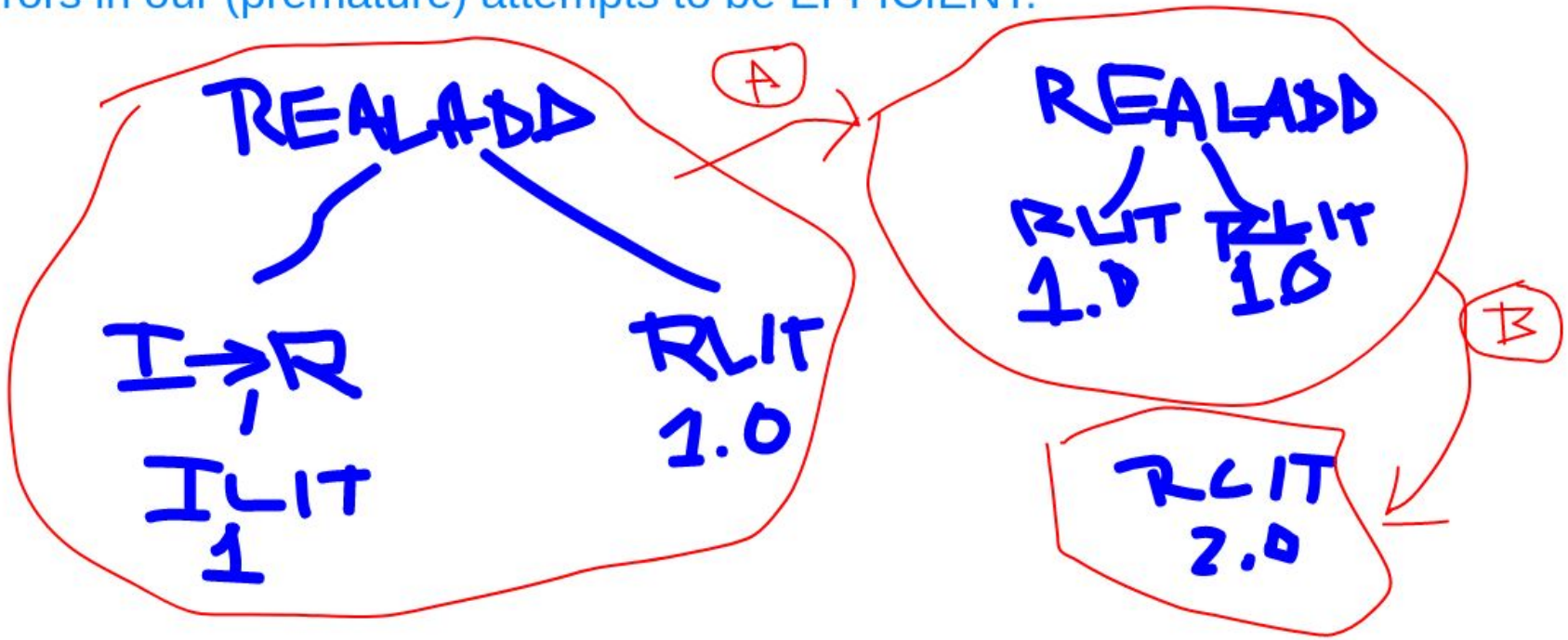

PLUSBINOP
1
1 + 1.0
2
INT_LIT
REAL_LIT


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
int_lit
1
realLit
1.0


REALADD
1
ILIT
1
RLIT
1.0
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

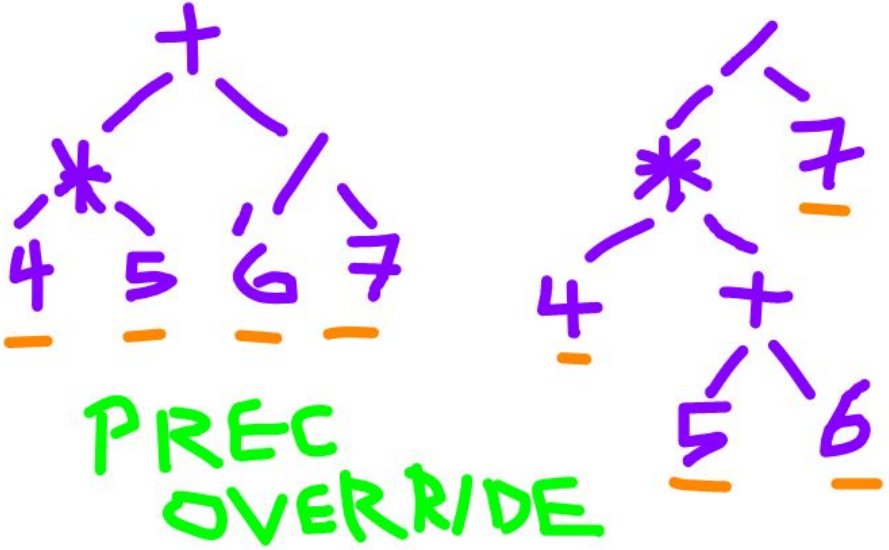
statementListWithReturn -->
    statementListWithReturn notReturnStmt

statementListWithReturn -->
    statementList returnStmt

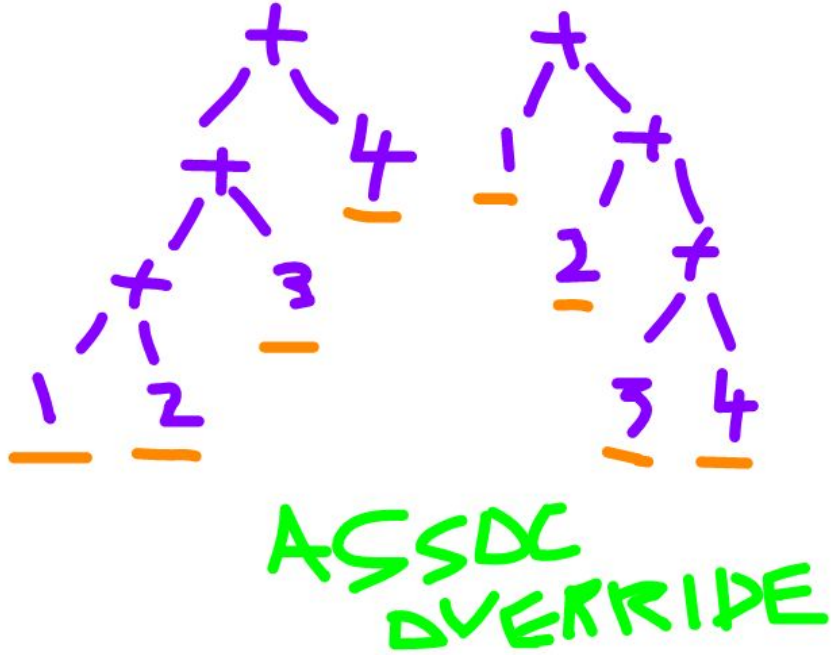
Semantic Checking of Expressions

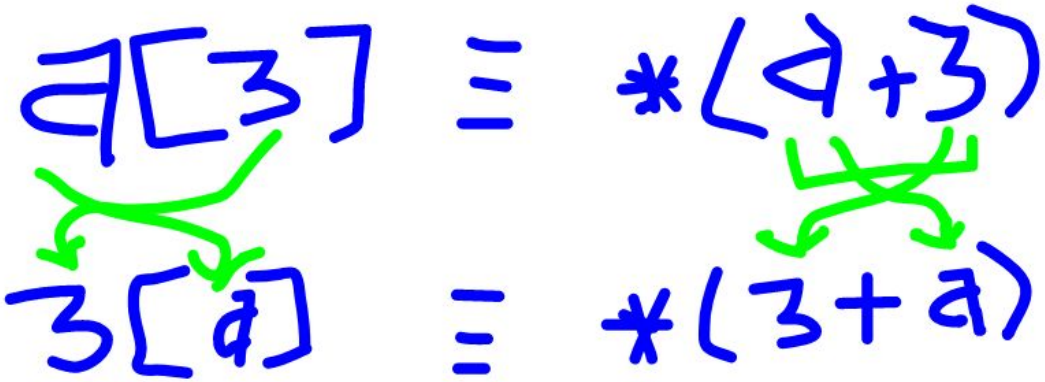
values and operators:

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?

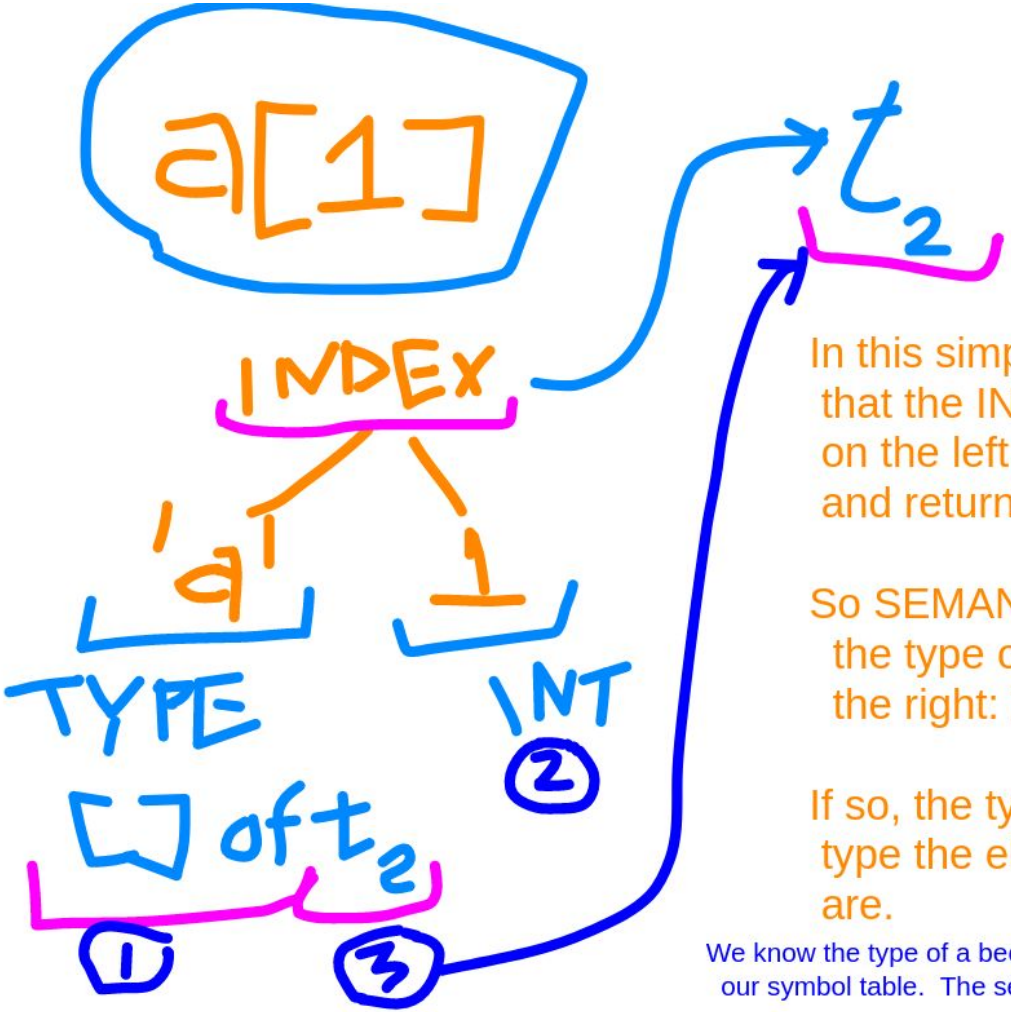
Yep, (int *).

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.

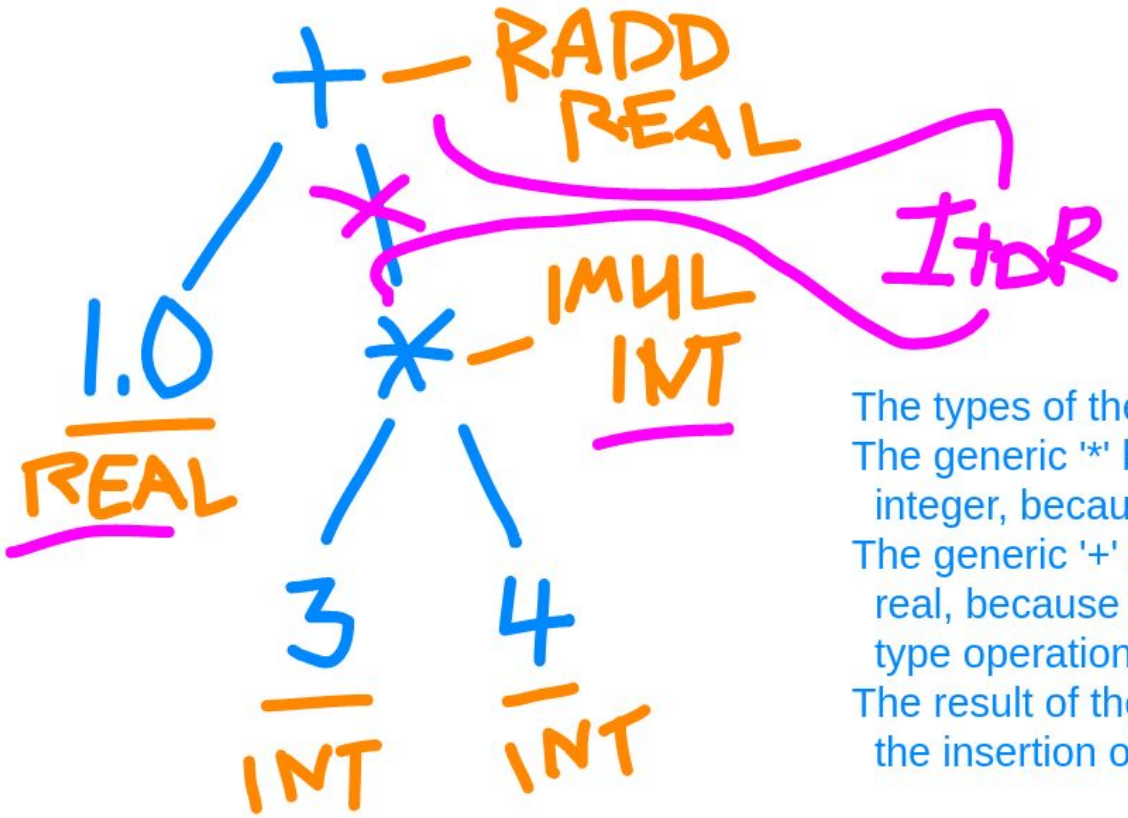


In this simple abstract case, we have said that the INDEX operator has a 1D array on the left and an expression on the right and returns the corresponding element.

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 '".

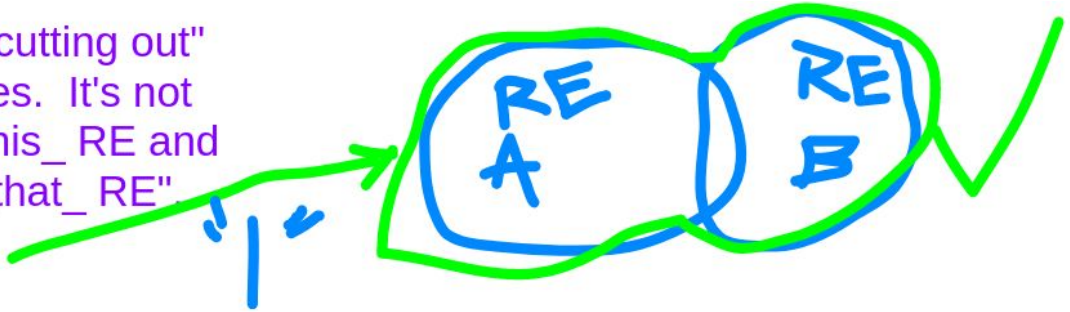
* flex does not care about (nor does it complain about) subsuming matches.

* 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.

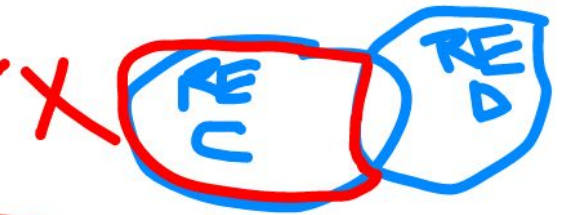
flex doesn't have the concept of "cutting out" except the [^] in character classes. It's not possible in flex to say "match _this_ RE and then _exclude_ what matches _that_ RE".

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.

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.

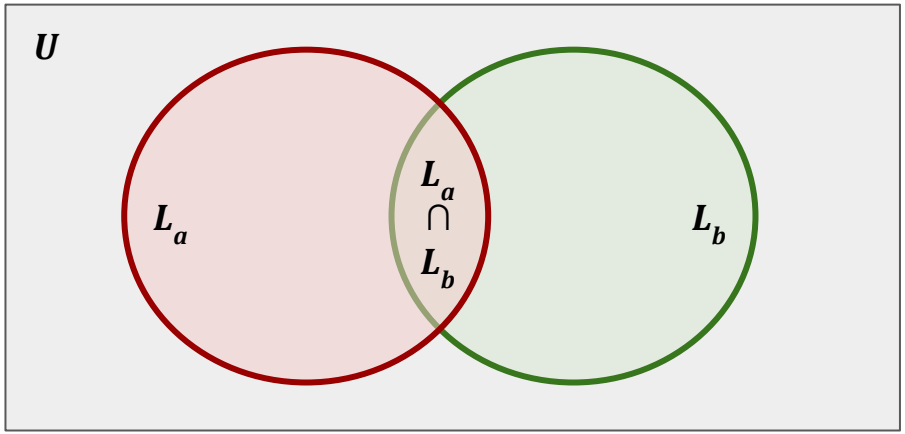
This is regular since we can take the first RE's finite automaton and change all of the accept states to reject states and vice-versa. Presto! We now have an FA that accepts the complement of the original language, thus it's regular. (Remember, we can convert REs to FAs and FAs to REs mechanically.)

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 $L_a - L_b$?

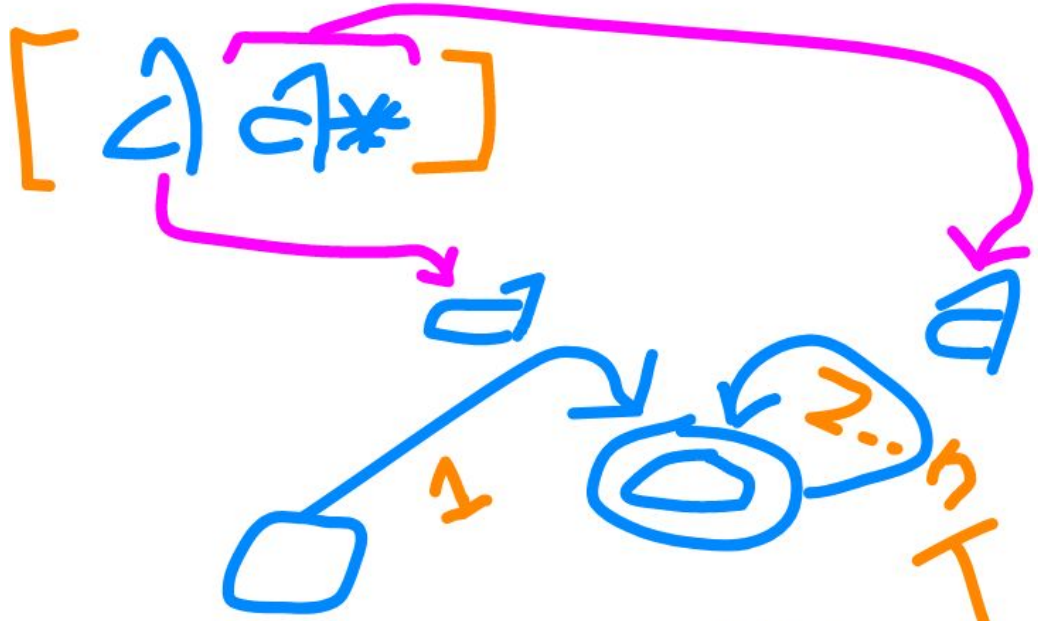
Well, once you understand how ¬L is regular if L is and $L_a \cap L_b$ is regular if L_a and L_b are, showing that $L_a - L_b$ is regular is easy.

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 ')'
```

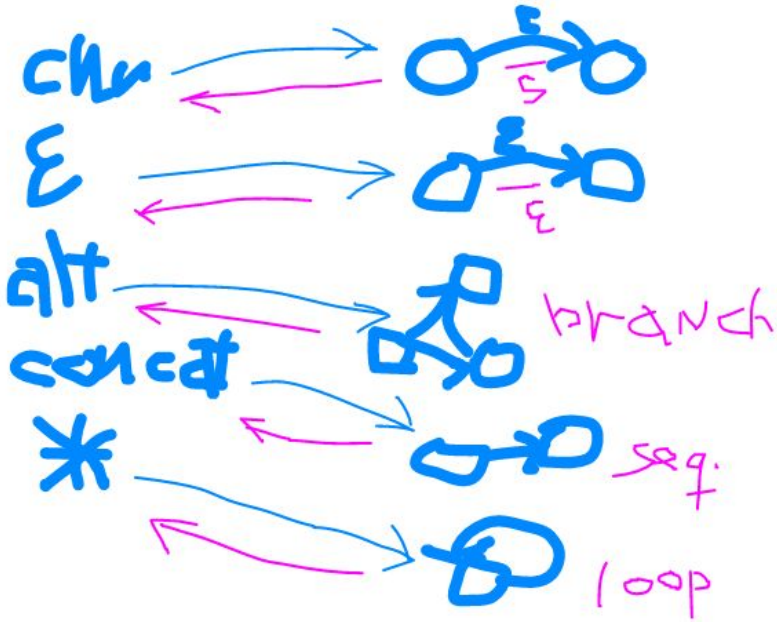
This sets the associativity by which side has the recursion (left or right) and the prec by the level of production rule. (The deeper the rule, the higher the precedence.)

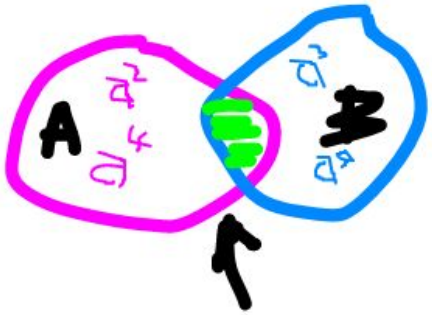
"REGULAR" language

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}, ...

$a^{z^n 3^m}$ $n,m \geq 1$

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

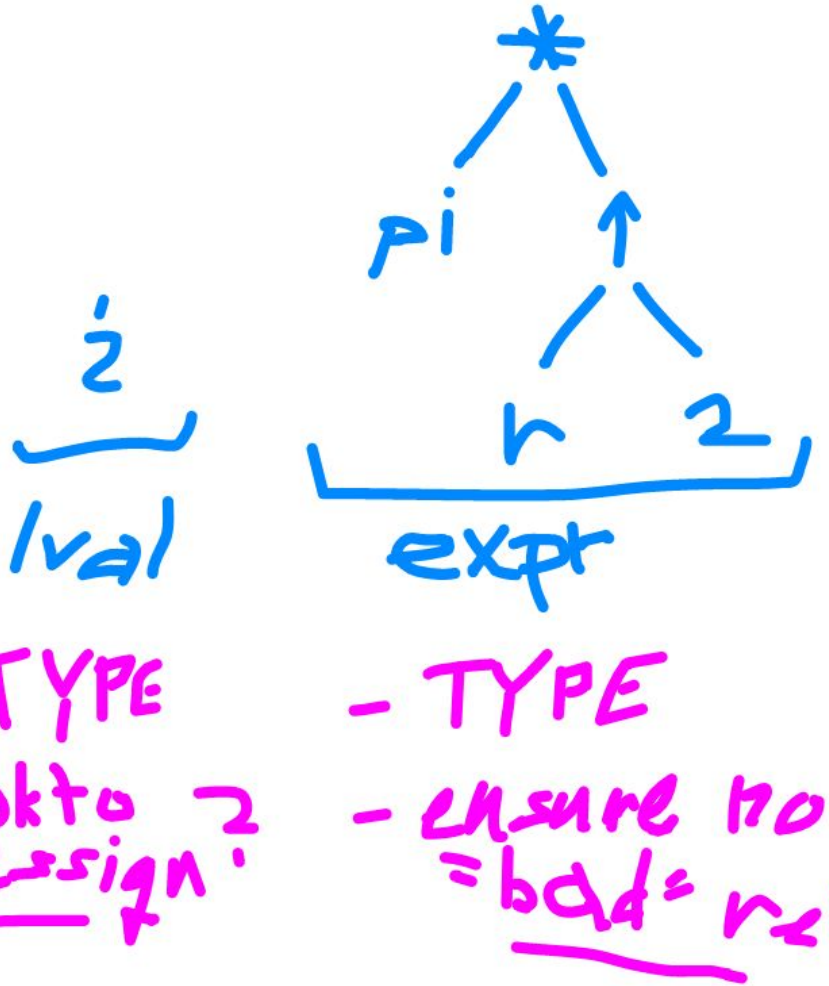
For example,
i <- pi*r^2

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).

EQ

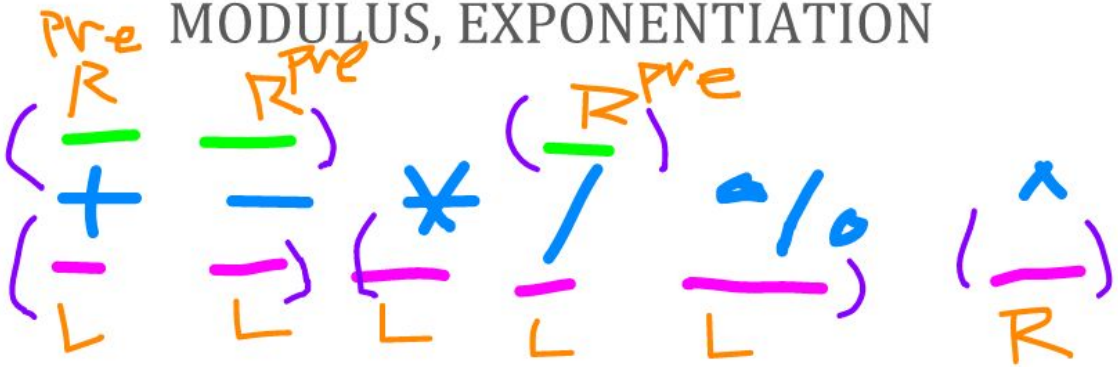
EQ and NE, obvious for ints/FP/string.
LT, LE, GT, GE obvious for ints/FP.
LT, LE, GT, GE are collating sequence for string. Case matters!

More detail req'd for mixing operand types.

Expressions: Operators

BINARY
WARY

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



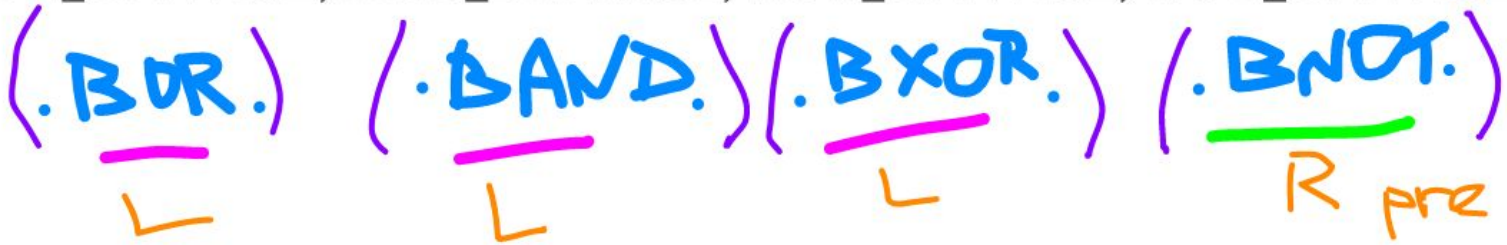
These numeric operators have the "normal" interpretations. If both operands are integers, the result is integer. If either (or both) operands are FP, the result is FP. Unary / is reciprocal: /2.0 -> 0.5. For unary operators, the result is the same as the operand, EXCEPT for unary / which is always FP.

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.

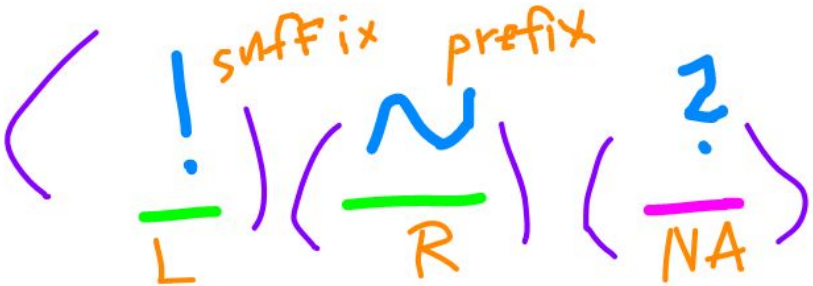
Each operator does the bit-by-bit operation on its operands.

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.

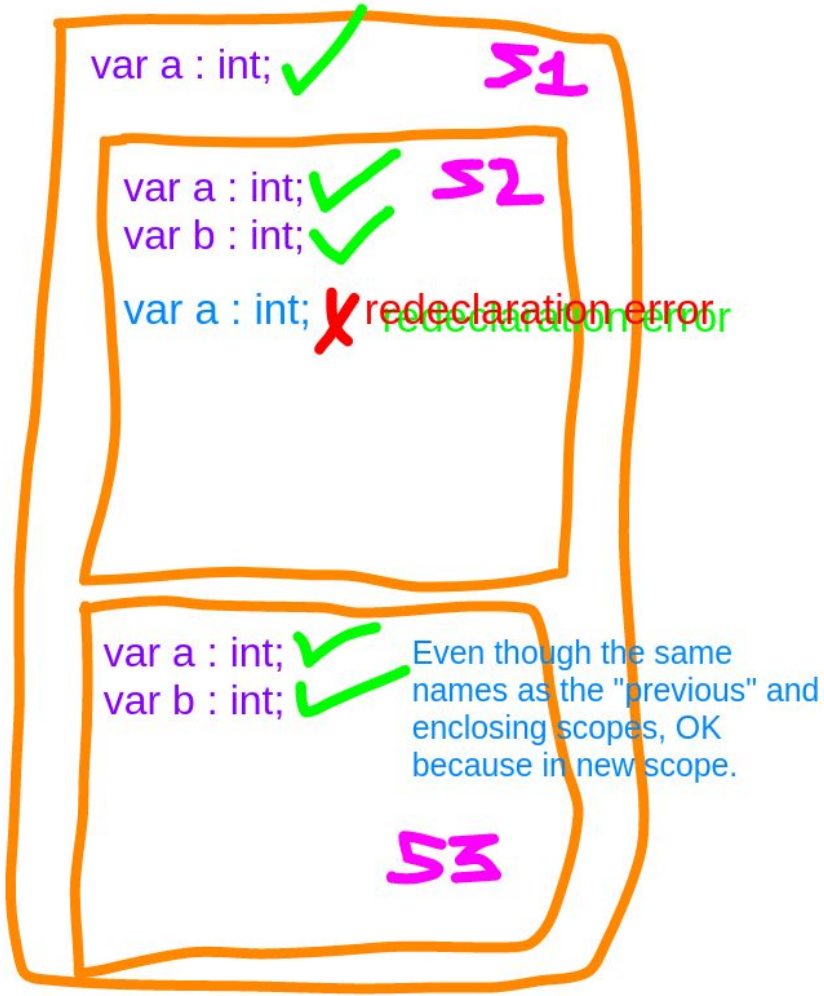
If not given, integers are initialized to 0, reals to 0.0, and strings to "".

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.)



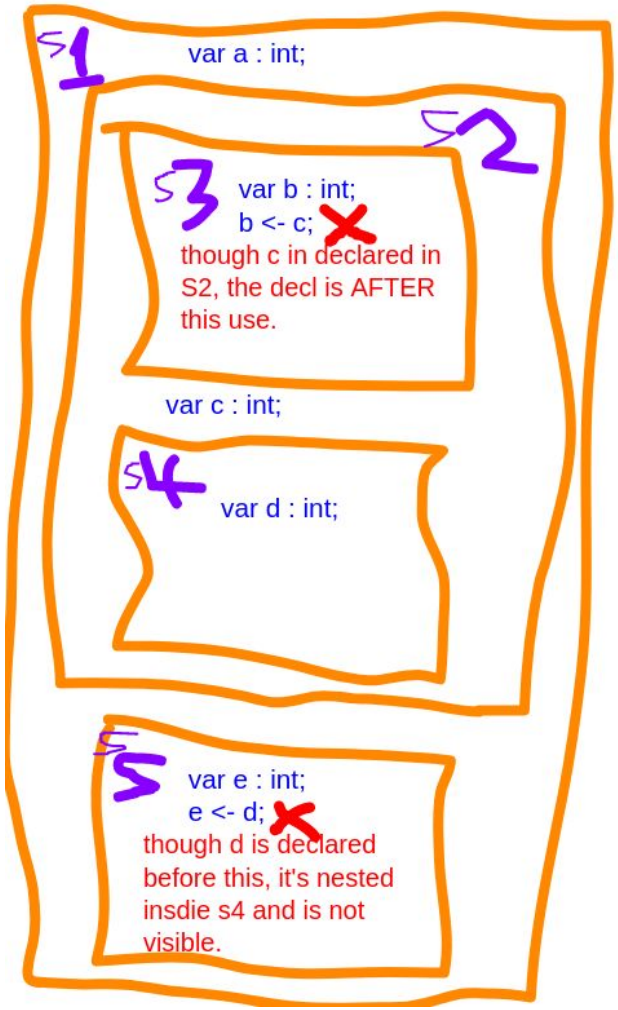
We need to be able to UNIQUELY identify any entry in the symbol table -- just a name is not sufficient. In the example, there are three (legal) "a"s and two "b"s.

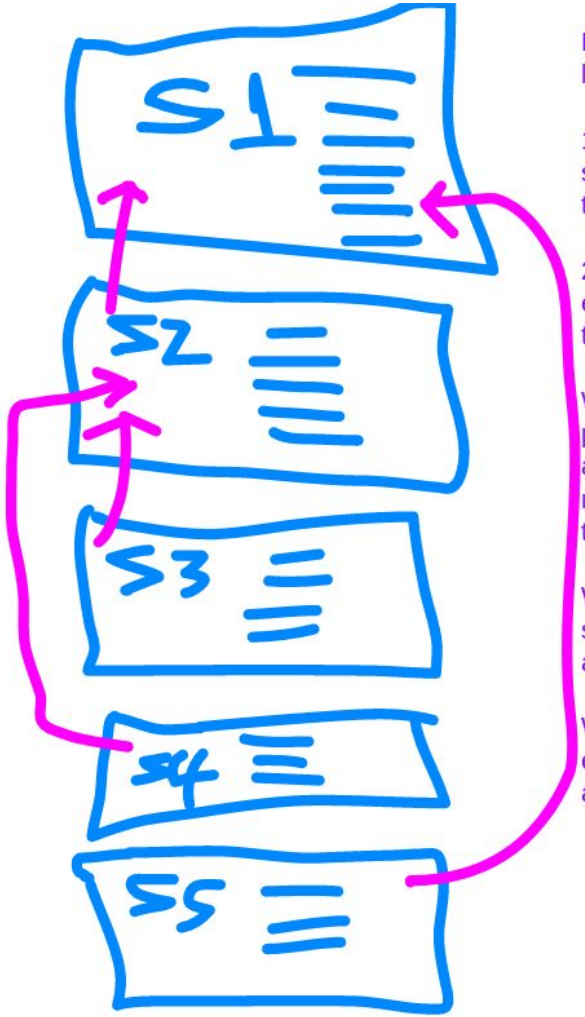
They are distinguishable because they are in different scopes.

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.

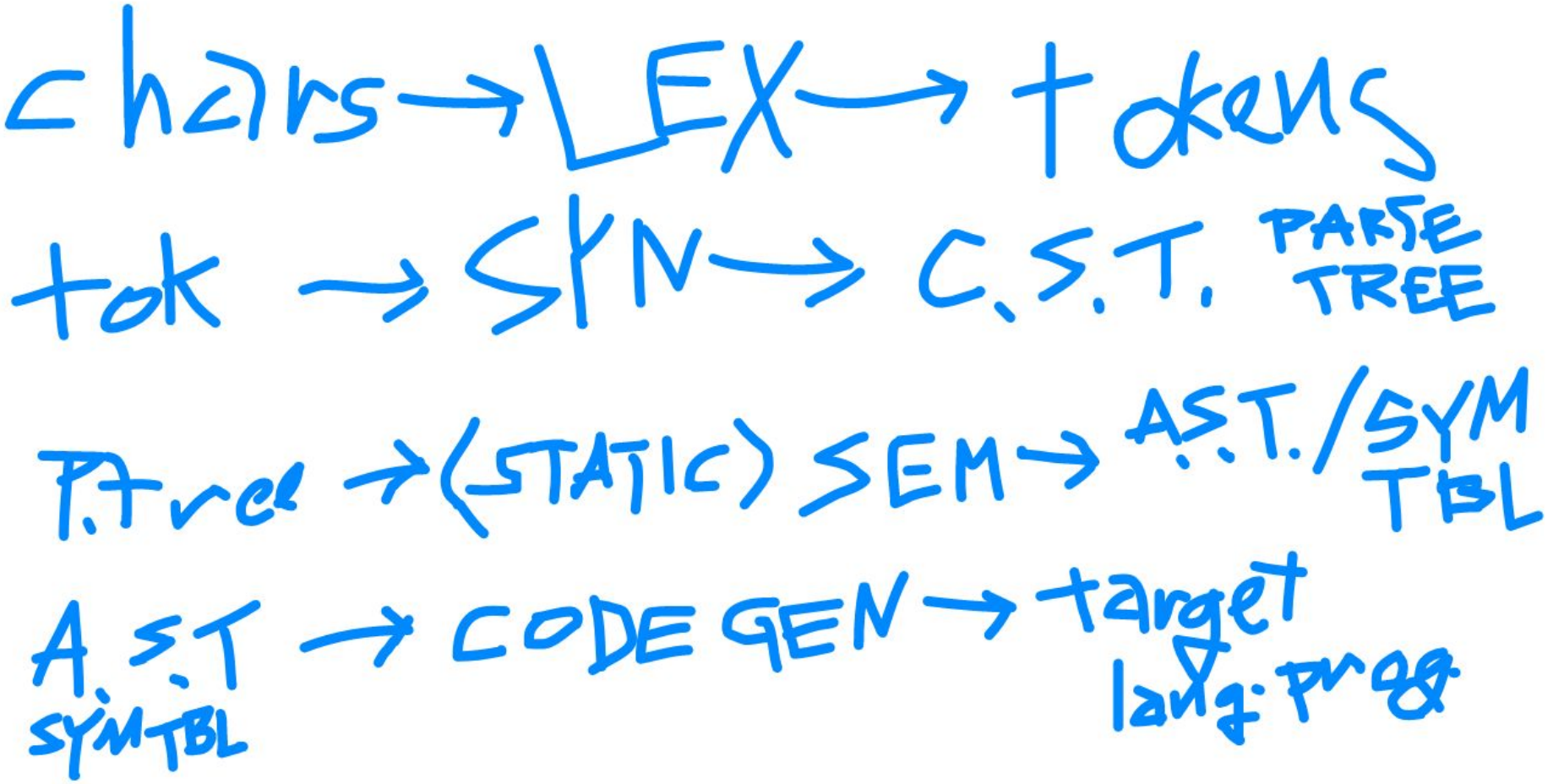
(2) The "body" (stmt_list) of a compound statement is a scope. This includes the if statement (the THEN clause, each of the ELIF clauses, and the ELSE clause).

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.

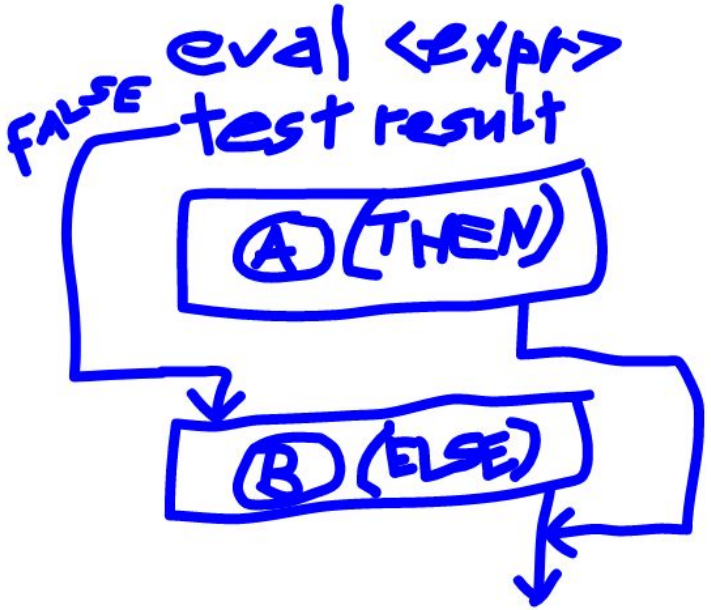
Code generation from a Pattern-based point of view: for a given construct in the source language (as processed by the semantic analysis phase) a pattern in the Target language is generated.

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

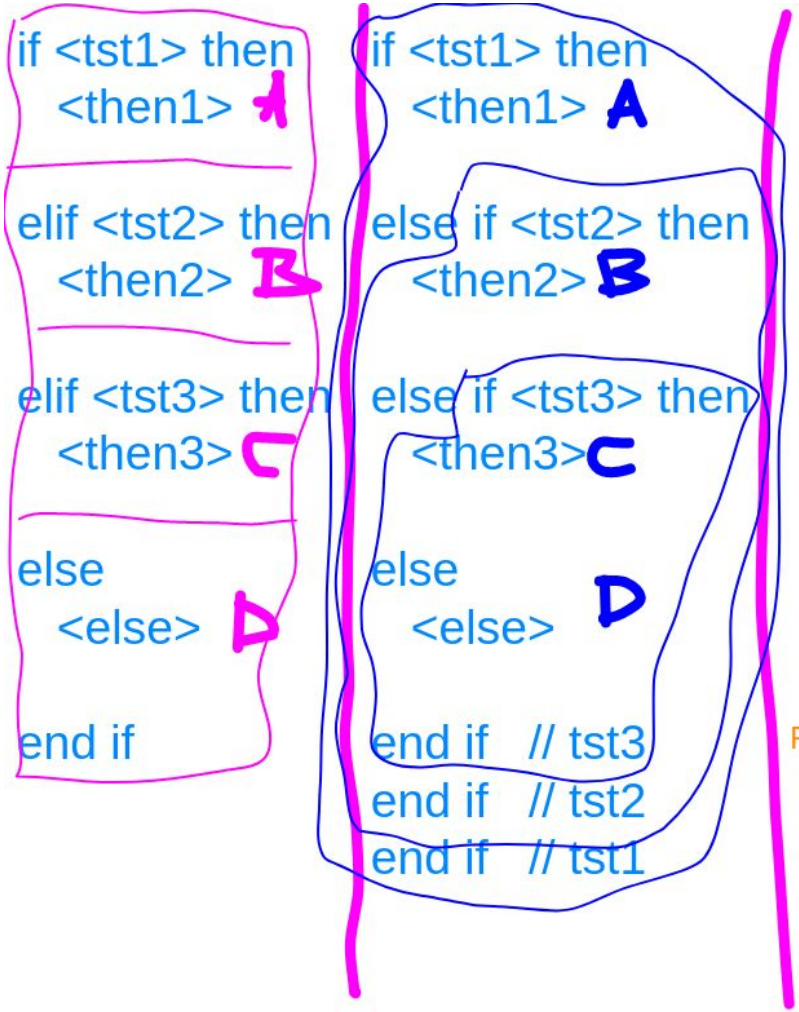


By "composition" we mean that a series of statements can be code generated by sequencing the code generation of each of the statements serially.

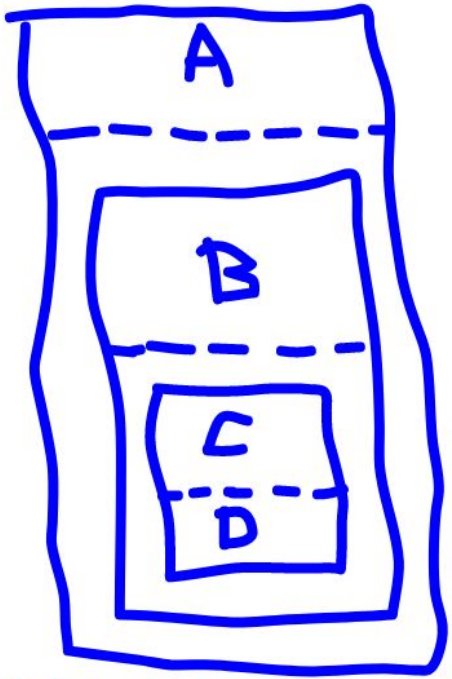
So codegen(A, B, C, D) ends up being codegen(A), codegen(B), codegen(C), codegen(D).

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).



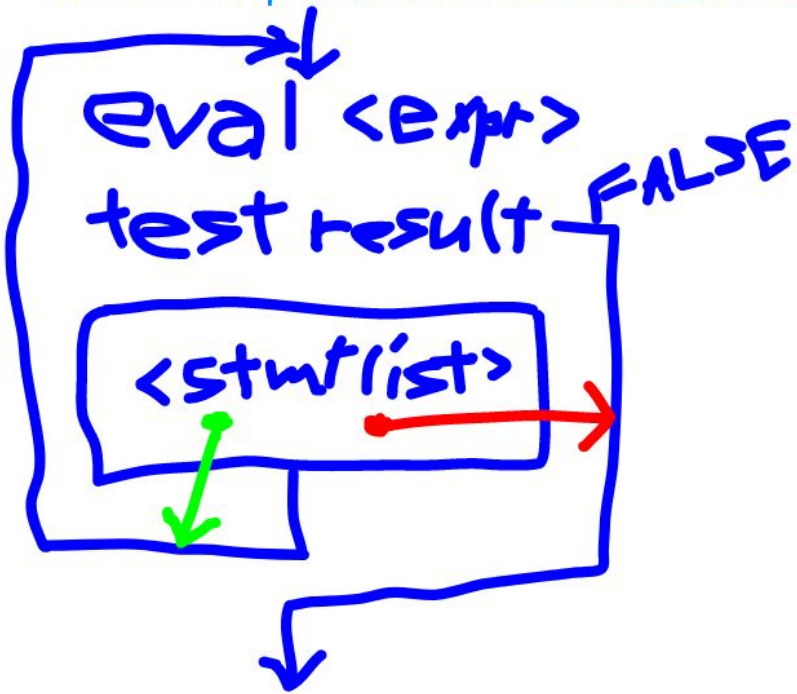
Requires code gen for an arbitrary number of ELIF clauses as well as an OPTIONAL ELSE clause.



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.

Consider the WHILE-DO statement:

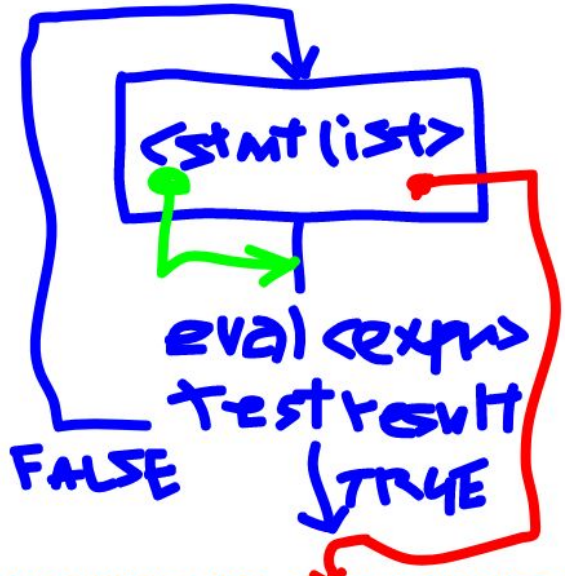
WHILE <expr> DO <stmtlist> END WHILE



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:

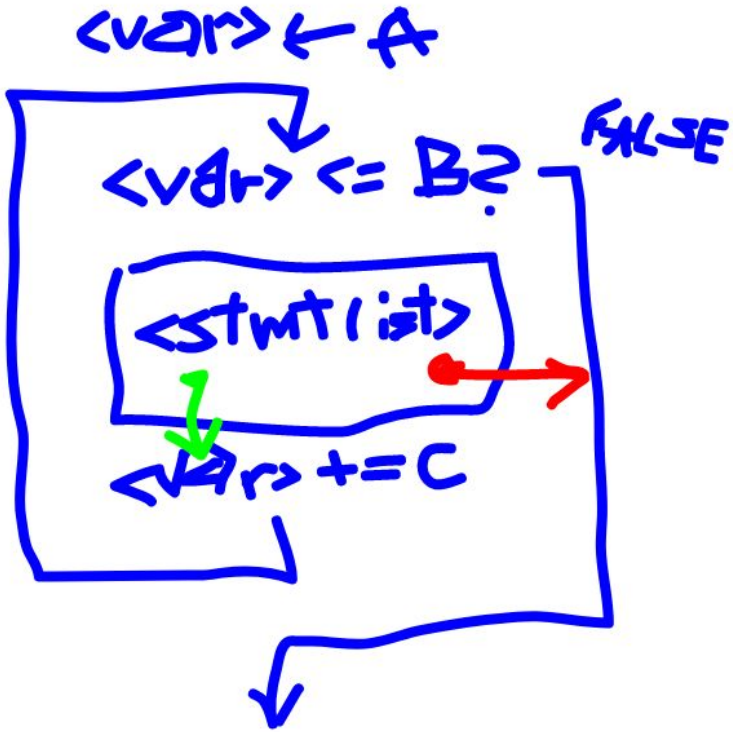
FOR var = <start> TO/DOWNTO <stop>
STEP <step> DO <stmtlist> END FOR

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,

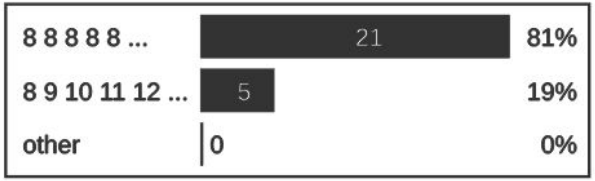
8 8 8 8 8 8 8 ...

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.

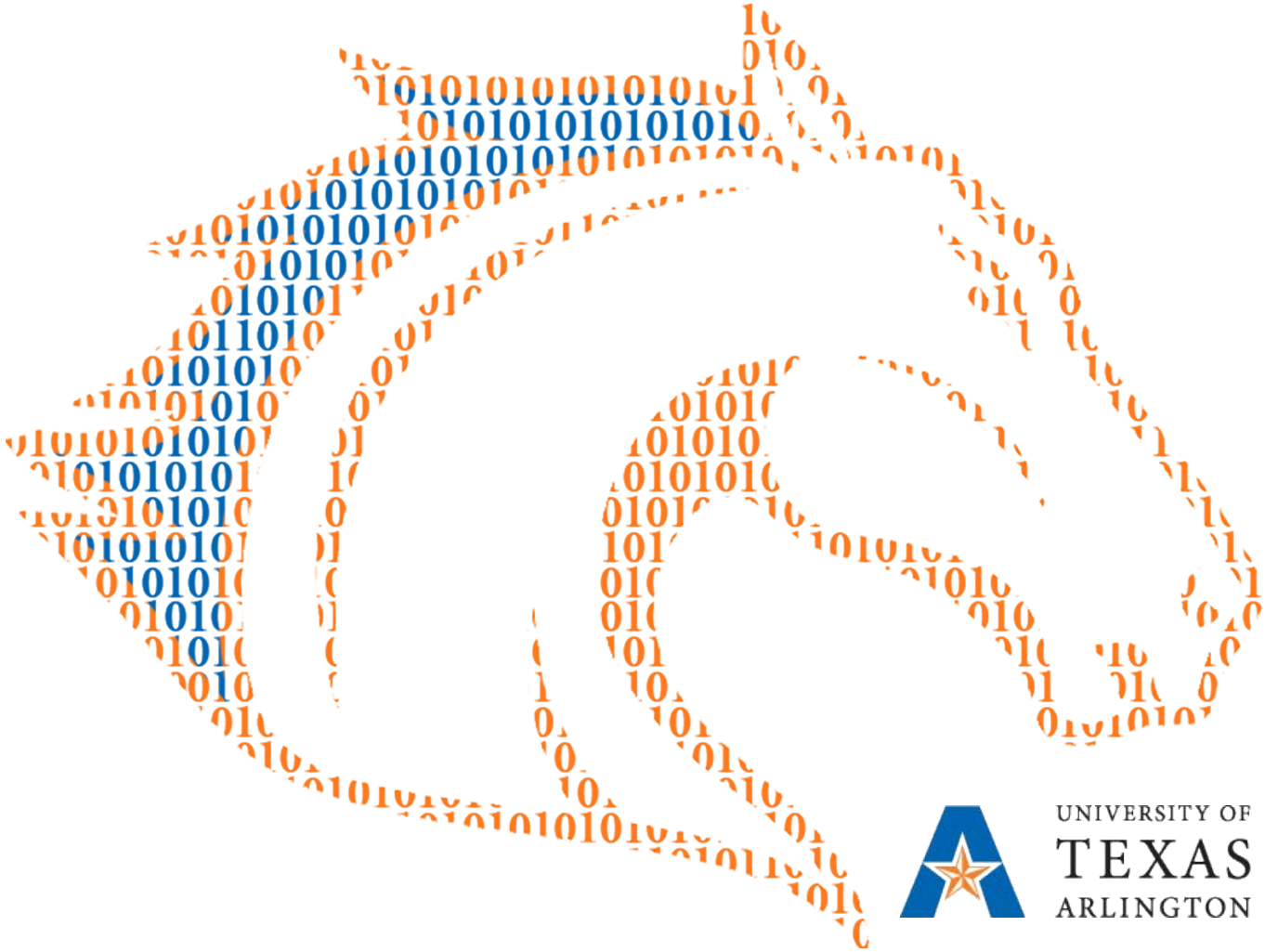
WRITE(<expr1>, <expr2>, ...)

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.

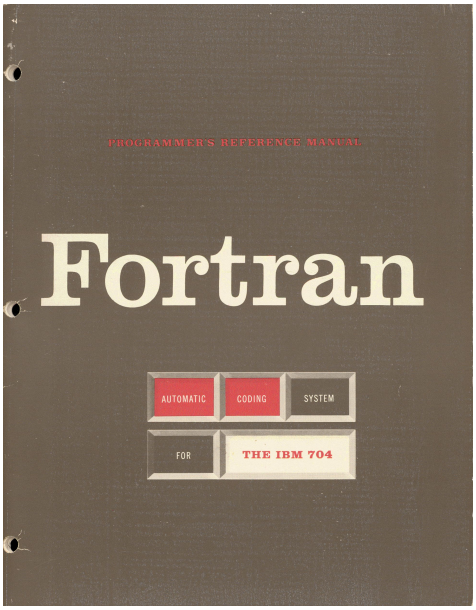


UNIVERSITY OF TEXAS ARLINGTON

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Compilers

CSE 4305 / CSE 5317
M00 Introduction
Fall 2020



M00

Introduction

First FORTRAN Manual, 1956 October 15

The IBM Mathematical Formula Translating System FORTRAN *is an automatic coding system for the IBM 704 EDPM. More precisely, it is a 704 program which accepts a* source program *written in a language – the* FORTRAN *language – closely resembling the ordinary language of mathematics, and which produces an* object program *in 704 machine language, ready to be run on a 704.*

FORTRAN *therefore in effect transforms the 704 into a machine with which communication can be made in a language more concise and more familiar than the 704 language itself. The result should be a considerable reduction in the training required to program, as well as in the time consumed in writing programs and eliminating their errors.*

Programming *with* Language

- ***Machine Code***
  - The actual ones and zeros that the processor uses to control its operation.
  - Each processor architecture has its own interpretation.
- Example is a GCD routine for the x86 architecture.

```
55 89 e5 53 83 ec 04 83 e4 f0
e8 31 00 00 00 89 c3 e8 2a 00
00 00 39 c3 74 10 8d b6 00 00
00 00 39 c3 7e 13 29 c3 39 c3
75 f6 89 1c 24 e8 6e 00 00 00
8b 5d fc c9 c3 29 d8 eb eb 90
```

Programming *with* Language

- ***Assembly Language***
  - Gives mnemonic (!) names to the instructions, registers, etc.
  - An *assembler* translates this into machine code.
  - Eventually included more than just 1-1 correspondence. (E.g., *macros*)
- Q: How was the first assembler written?

```
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    andl  $-16, %esp
    call  getint
    movl  %eax, %ebx
    call  getint
    cmpl  %eax, %ebx
    je    C
A:  cmpl  %eax, %ebx
    jle   D
    subl  %eax, %ebx
B:  cmpl  %eax, %ebx
    jne   A
C:  movl  %ebx, (%esp)
    call  putint
    movl  -4(%ebp), %ebx
    leave
    ret
D:  subl  %ebx, %eax
    jmp   B
```

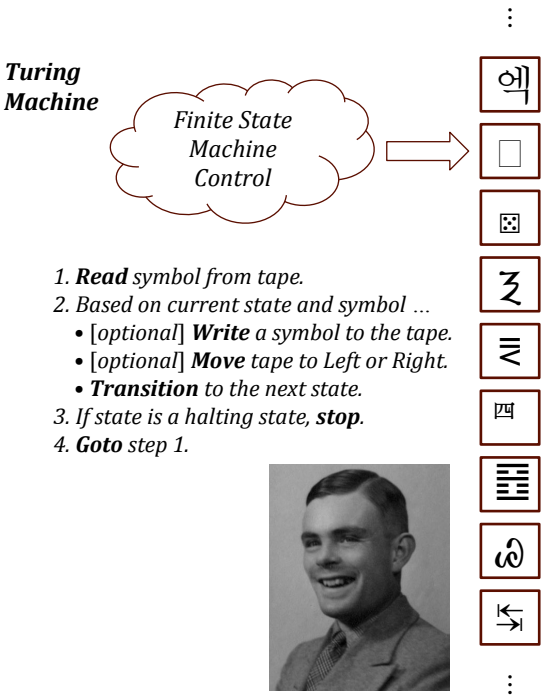
Programming *with* Language

- ***High-Level Language***
  - Gets away from any particular processor architecture. (Generally …)
  - A *compiler* translates the high-level language to (assembly which is then translated to) machine code. (Generally …)
  - At first, humans could write better assembly code than the compiler generated, so slow to catch on.
  - But, reduced the number of statements that had to be written by a factor of 20!
  - It took 18 staff-years (!) to write the first FORTRAN compiler.
- Q: How was the first compiler written?

```
      FUNCTION IGCD()
      READ(5,500) IA, IB
500   FORMAT(2I5)
600   IF (IA.EQ.IB) GOTO 800
      IF (IA.GT.IB) GOTO 700
      IB = IB - IA
      GOTO 600
700   IA = IA - IB
      GOTO 600
800   IGCD = IA
      RETURN
      END(2, 2, 2, 2, 2)
```

[*Turing Complete*]

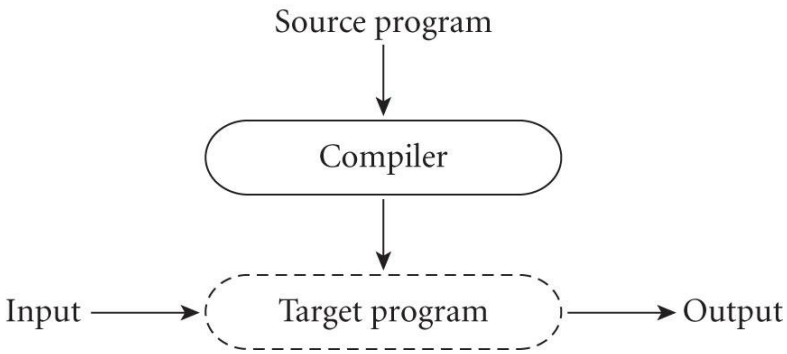
- ***Informally***, any real-world general-purpose computer or computer language can approximately simulate the *computational* aspects of any other real-world general-purpose computer or computer language.
- In other words, they are all the same; it's just that some may be more or less *convenient* for any particular *computation*.



http://www.turingarchive.org/browse.php/K/7/9-16

Compiling a Program

- A *compiler* translates a *source* program into an equivalent *target* program and then goes away. At some later time, the target program can be executed.

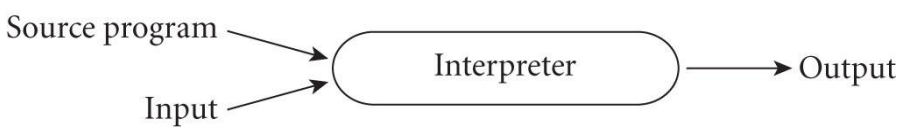


Compiling a Program

- During compilation, the compiler is the *locus of control*.
- During its own execution, the target program is the locus of control.
- Typically, the compiler is a machine language program.
- Typically, the target program is also a machine language program.
- Generally leads to best performance.
- *Translate once, run many times*.

Interpreting a Program

- An *interpreter*, on the other hand, typically stays around for the execution of the target program.
  - The interpreter is the locus of control during all parts of the execution.
  - The interpreter is in effect a *virtual machine* whose machine language is the programming language.
  - *Translate every time*.



Interpreting a Program

- Generally, interpretation is more flexible and has better diagnostics than compilation.
  - The source code of the program is still available.
- Some language features are very difficult to implement without interpretation.
  - "On-the-fly" code generation

[*Read-Eval-Print Loop (REPL)*]

- A *Read-Eval-Print Loop* is an interactive environment wherein the user types a line at a time which is then evaluated and results displayed.
  - *Read* — Get input from user.
  - *Eval* — Determine value.
  - *Print* — Display result to user.
  - *Loop* — Do again, until terminated.
- Can be created for any text-based language, Lisp, Python, Java …
- REPL is a one-liner in Lisp for Lisp:

  `(loop (print (eval (read))))`
- Lot of REPLs available *live* and *on-line*:

  `http://joel.franusic.com/Online-REPs-and-REPLs/`

```
(base) dalioba@achpiel:~$ python
Python 3.7.4 (default, Aug 13 2019, 20:35:49)
[GCC 7.3.0] :: Anaconda, Inc. on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 1 + 2
3
>>> a = 10
>>> b = 5
>>> a + b
15
>>> c = a + b
>>> print( c )
15
>>> c
15
>>> b = "Hi, there!"
>>> c = a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> for i in range( 5 ) :
...   print( i, i*i, i*i*i )
...
0 0 0
1 1 1
2 4 8
3 9 27
4 16 64
>>> quit()
(base) dalioba@achpiel:~$
```

Compilation vs. Interpretation

- In both cases, instructions are executed on the target processor.
- A compiler generates instructions by analyzing the source code in its *entirety* and optimizes the generated code based on the *entire source code* and specific optimizations.
- The generation of the instructions is (generally) independent of the execution of the target program.

Compilation vs. Interpretation

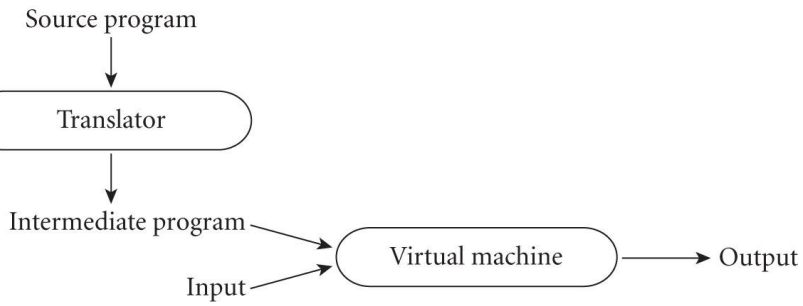
- An interpreter typically reads one “line” at a time.
  - It cannot perform overall analysis of the code as it does not know all of the code.
  - It therefore executes one line at a time without optimization.
- It must read the source code and generate instructions as it is executing the target program.
- It may take several operations for an interpreter to accomplish the same operation a compiler would have generated one machine instruction for.

Compilation vs. Interpretation

- Interpretation
  - Don't have to wait for compile and link steps.
  - Usually takes less space than a compiler.
  - Much easier to port an interpreter to a new architecture.
- Compilation
  - Generally much better performance.
  - Generally catches many errors earlier, before the target program even executes.

Compiling *and* Interpreting a Program

- While compilation and interpretation are different, they can be used together.



Compiling *and* Interpreting a Program

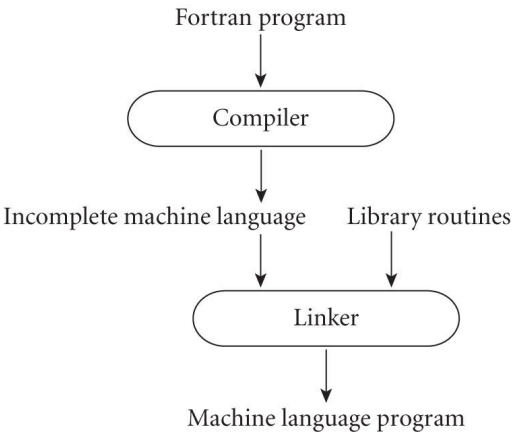
- If the initial translator is "simple", we'd still call this interpretation. If it's "complex" we'd call this compilation.
  - Subjective!
  - Also, both parts can be quite complex, as in the case of Java.
- Instead, we'll use "compiling" to mean that a *complete analysis* of the source code is done by the translator.
  - Not just a "mechanical" transformation, as, e.g., cpp does.
- Also, the intermediate program would not bear a strong resemblance to the source code. It's a *nontrivial* translation.

Pure Interpretation

- The earliest implementations of Basic were close to pure interpreters.
  - The original characters were read and reread and reread as the source code was interpreted. Removing comments sped up the program's execution!
- Generally, a modern interpreter processes the source code to remove whitespace and comments, group characters into *tokens*, and even perhaps identify syntactic structures.

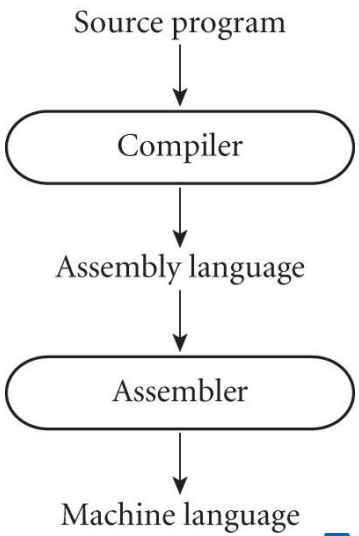
Pure Compilation

- (Early) Fortran implementations however were close to pure compilation.
  - The source code is translated into machine code.
  - May have a *library* of subroutines for shared use.
  - A *linker* puts it all together.
- There's still *some* interpretation.
  - `FORMAT` statements



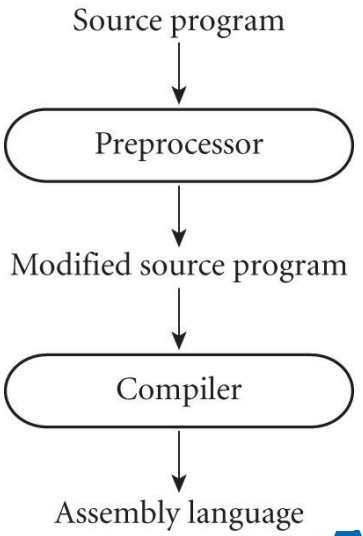
Chain of Compilation

- Many compilers generate assembly language instead of machine code.
  - Can make debugging easier
  - Helps isolate compiler from changes in the operating system



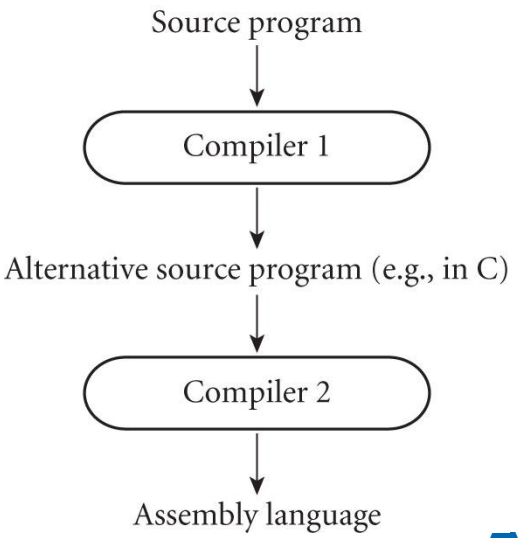
Chain of Compilation

- Many compilers (e.g., C) begin with a *textual* preprocessor that,
  - Removes comments
  - Expands *macros* (`#define`)
  - Provides *insertion* (`#include`)
  - Provides *conditional compilation* (`#if`)



Chain of Compilation

- Some compilers generate a relatively high-level intermediate program and then use *another* compiler to get to the final target program.
  - Called *source-to-source* compiling, C++ was originally implemented this way.
  - Still considered a true compiler!
    - Full analysis, non-trivial transform



Self-Hosting Compiler

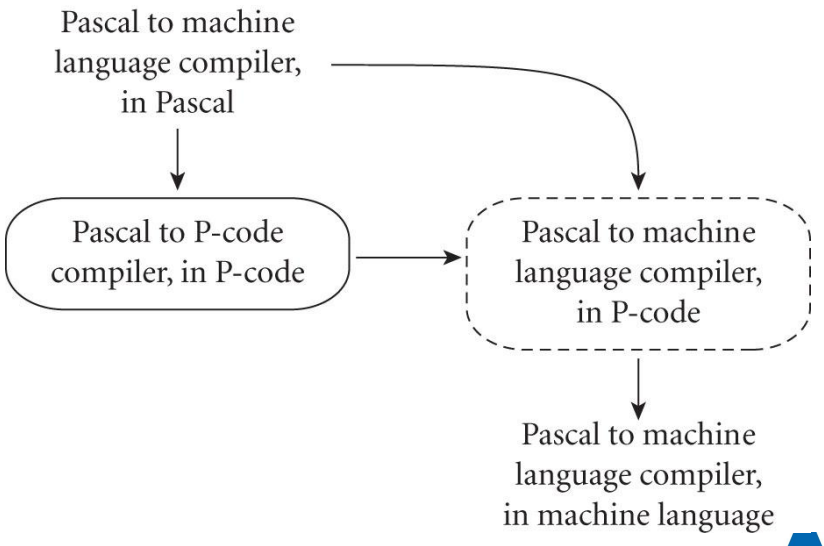
- A *self-hosting* compiler is written in its own language.
  - An Ada compiler written in Ada, a C compiler written in C, and so forth.
- So how to compile it the first time?
  - *Bootstrapping*: Start with a very simple implementation that knows just enough to get to the next level.
    - Often the earliest parts are *interpreters*.
  - Each step up adds more capability until the entire compiler can be compiled.

Self-Hosting Compiler Example

The original Pascal distribution included:

- A Pascal compiler, written in Pascal, that generates *P-code*.
- The same compiler, already in P-code.
- A P-code interpreter written in Pascal.

So, translate the P-code interpreter into a local language, then use it to run the P-code version of the compiler, then compile the Pascal version of the compiler.



[*P-Code*]

- A “P-Code” is a very simple language that is easy to translate to machine code.
  - “P” for *portable* or *pseudo*.
  - Used to make it easier to port software from one machine to another.
  - Also used to isolate compiler front ends from back ends.
- P-Code can be considered an intermediate form in the compilation process.

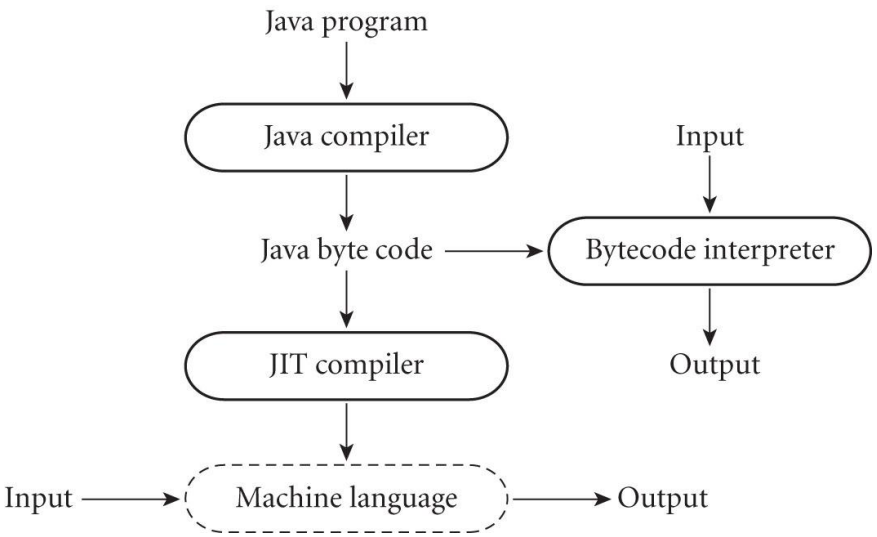
Compiling Interpreted Languages

- Some programs in traditionally interpreted languages can be compiled under a set of assumptions (about, e.g., types, bindings).
  - If at run-time the assumptions are violated, the program drops back into interpreted mode.
  - Otherwise, the performance advantages of compiled code can be obtained.

Just-In-Time Compilation

- An extension of this concept is *JIT*.
- Compilation can be *delayed* until the last possible moment.
- Allows for handling "on-the-fly" code, optimization based on run-time conditions, etc.
- Also supports platform-independent intermediate representation that can be distributed and then compiled locally into machine code.
- Examples include Java, C#.

Just-In-Time Compilation



Overview of Compilation

- Compilation is one of the most intensely studied aspects of computer science.
- Much of compilation has been studied so much that what used to be very hard is now fairly straightforward (not to say *easy*).
  - The first Fortran compiler cost 18 staff-years.
  - Now, writing a "compiler" is a semester project for an undergraduate.

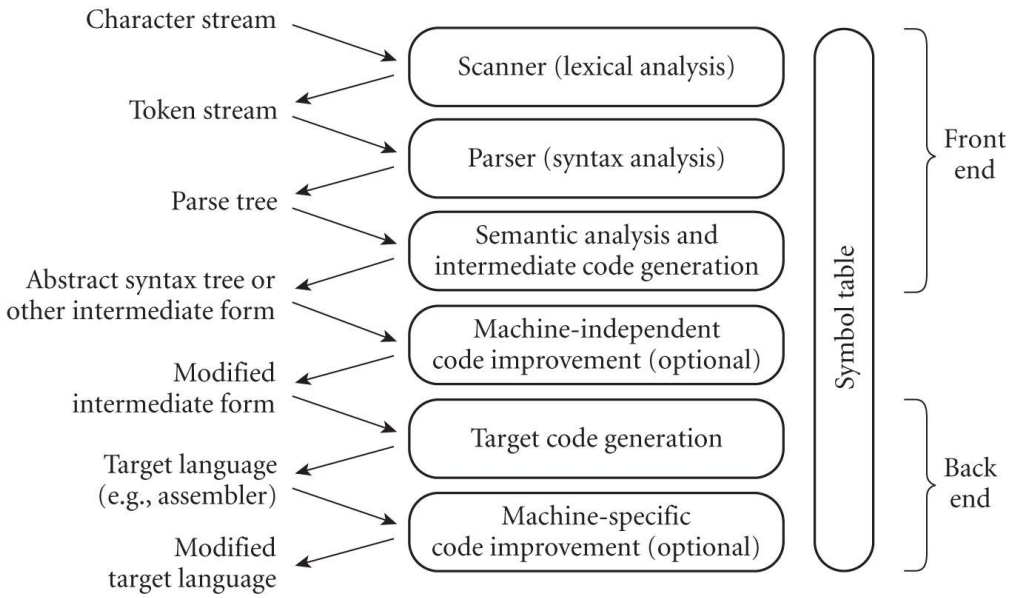
Overview of Compilation

- The compilation process is divided into *phases*
  - Each phase determines new information to be used later or transforms what has already been learned for later use.
  - The first phases analyze the source program to ***discover its meaning***. This is the compiler *Front End*.
  - The last phases ***construct*** (and possibly improve) ***the target program***. This is the compiler *Back End*.

Overview of Compilation

- One may hear of compiler *passes*
  - A *pass* is a phase or set of phases that are serialized with the respect to the others.
  - It does not start until all previous phases / passes have completed and it runs to completion before subsequent phases / passes start.
  - Sometimes a pass may be an entirely separate program, reading a description from and writing one to files.
  - Passes can be used to share parts of the compilation process.

Phases of Compilation



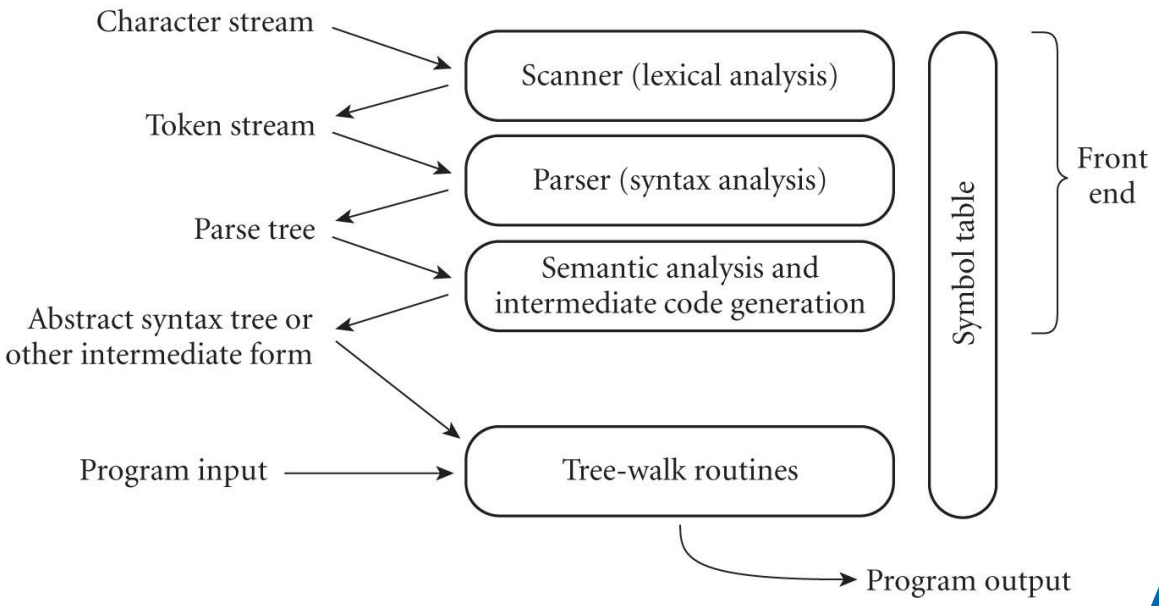
Front End vs Back End

- The *Front End* of a compiler (or interpreter) comprises the lexical, syntactic and semantic analysis steps along with the generation of intermediate code.
- The *Back End* of a compiler comprises the optimization of the intermediate code, target code generation, and machine-specific code optimization steps.
  - Interpreters don't have back ends.
    - (At least not the way compilers do.)
- The front end is concerned with determining what the program ***means***. The back end with ***producing the corresponding target code***.

Overview of Interpretation

- An interpreter shares the general structure of a compiler up through the Front End.
- However, instead of generating a target program, the interpreter "executes" the intermediate form directly.
  - This execution is commonly accomplished by a set of mutually-recursive routines that traverse ("walk") the abstract syntax tree, executing its nodes in order.
  - Another technique is to generate *bytecode* and hand it off to a bytecode engine. (This is getting closer to compilation.)

Phases of Interpretation



Overview of Compilation

- Lexical Analysis (*Scanning*)
  - The *scanner* (or *lexer*) reads individual characters and groups them into *tokens*.
  - These tokens are the smallest meaningful units of a program.

```
int main() {
  int i = getint(), j = getint();
  while (i != j) {
    if (i > j) i = i - j;
    else j = j - i;
  }
  putint(i);
}
```

```
int     main  (     )  {   int     i       =
getint  (     )     ,  j   =       getint  (
)       ;     while (  i   !=      j       )
{       if    (     i  >   j       )       i
=       i     -     j  ;   else    j       =
j       -     i     ;  }   putint  (       i
)       ;     }
```

Overview of Compilation

- Syntactic Analysis (*Parsing*)
  - Organizes the tokens into a *parse tree* that hierarchically represents higher-level constructs in terms of their lower-level parts.
  - Each construct is a *node*, its parts are its *children*, the *leaves* are the tokens from the lexical analysis phase.
  - The tree is constructed from the tokens by means of a set of recursive rules known as a *context-free grammar*.
  - The grammar defines the *syntax* of the language.

Overview of Compilation

- Context-free grammar excerpt, from C.
  - Shows the syntax of the `while` statement.
  - ε represents the empty string.
- Quite complicated!
- In general, parse trees have an insanely immense amount of information, even for relatively small programs.

iteration-statement ⟶ `while` (*expression*) *statement*

statement ⟶ *compound-statement*
compound-statement ⟶ { *block-item-list_opt* }

block-item-list_opt ⟶ *block-item-list*
block-item-list_opt ⟶ ε

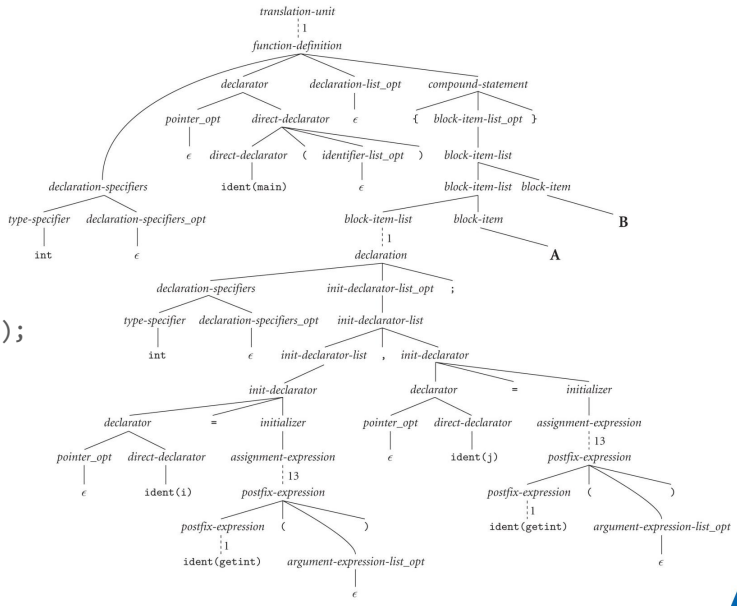
block-item-list ⟶ *block-item*
block-item-list ⟶ *block-item-list block-item*
block-item ⟶ *declaration*
block-item ⟶ *statement*

http://www.quut.com/c/ANSI-C-grammar-y.html

Overview of Compilation

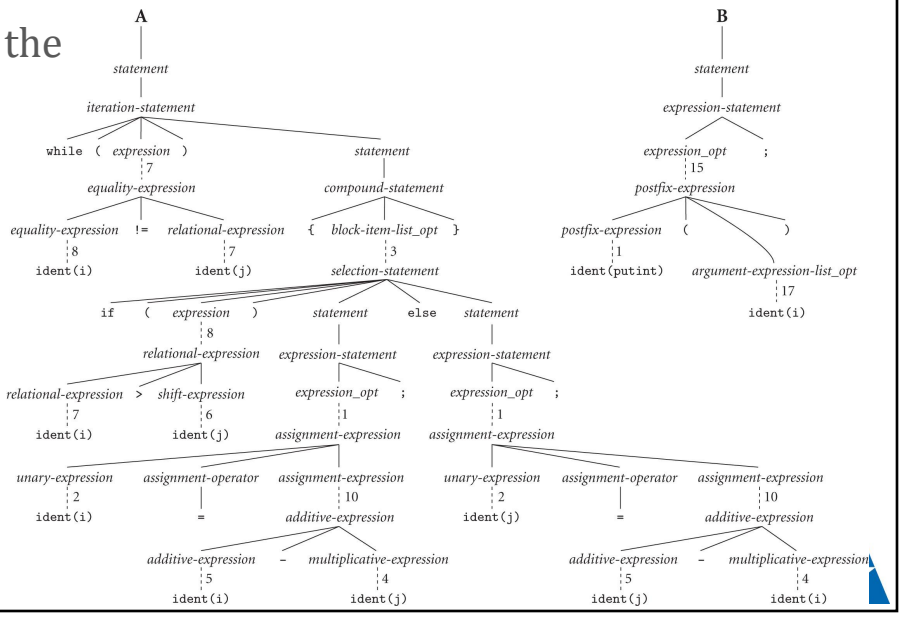
- Parse Tree for the GCD example (Part 1)

```
int main() {
  int i = getint(), j = getint();
  while (i != j) {
    if (i > j) i = i - j;
    else j = j - i;
  }
  putint(i);
}
```



Overview of Compilation

- Parse Tree for the GCD example (Part 2)



Overview of Compilation

- Semantic Analysis
  - Discovers the *meaning* of the source program. For example,
    - Determines when the use of the same identifier means the same program entity.
    - Ensures uses are consistent, both as a test of the legality of the source program and to guide generation of code in the back end.
  - Builds and maintains a *symbol table* to map each identifier to the information known about it, including
    - Type, internal structure (if any), scope, ...

Overview of Compilation

- Semantic Analysis
  - Enforces a large variety of rules that are not captured by the syntactic structure of the program, for example (in C),
    - Declaration of identifier before use
    - No use of identifier in an inappropriate context
    - Correct number / type of parameters in subroutine calls
    - Distinct, constant labels on the branches of a switch statement
    - Non-`void` return type function returns a value explicitly
  - These example rules are *static semantic* checks as they depend on only the structure of the program as it is written and can be enforced at compile time.

Overview of Compilation

- Semantic rules that can't be enforced until runtime are *dynamic semantics*, for example,
  - Variables are not used in an expression unless they have been assigned a value.
  - Pointers are not dereferenced unless they refer to a valid address.
  - Array subscripts are within bounds.
  - Arithmetic expressions do not overflow.

Overview of Compilation

- Those dynamic semantics rules just given do *not* apply to C.
  - It would have saved a lot of trouble and debugging time if they were automatically enforced.
  - Q: Why aren't they?

- C has very little in the way of dynamic semantics rule enforcement.
  - As in … *none*. (Is that true?)

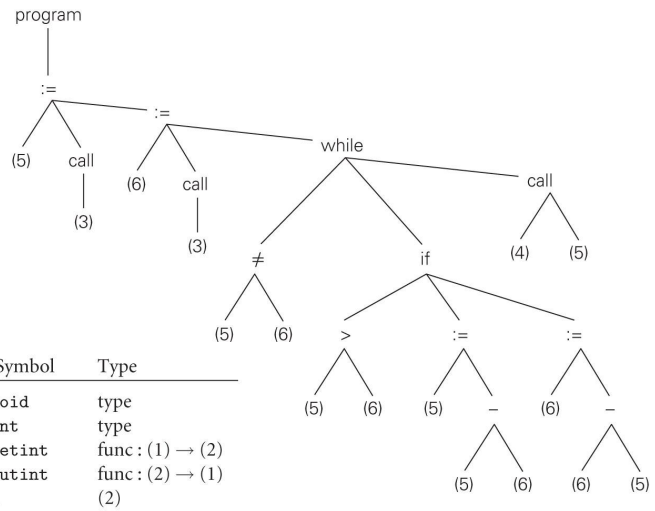
Overview of Compilation

- The parse tree is sometimes known as a *concrete syntax tree* because it completely shows how a sequence of tokens was derived using the CFG (Context-Free Grammar).
- As the static semantics phase runs, it transforms the parse tree to an *abstract syntax tree (AST)* which retains only the essential information.
- Annotations are also made with information useful to the rest of the process.
- Nodes get *attributes* added to them as required.

Overview of Compilation

- Abstract Syntax Tree and symbol table for GCD program

```
int main() {
  int i = getint(), j = getint();
  while (i != j) {
    if (i > j) i = i - j;
    else j = j - i;
  }
  putint(i);
}
```



Index	Symbol	Type
1	`void`	type
2	`int`	type
3	`getint`	func : (1) → (2)
4	`putint`	func : (2) → (1)
5	`i`	(2)
6	`j`	(2)

Overview of Compilation

- It's common for an interpreter to use the AST as its representation of the program to run.
- "Execution" then amounts to a traversal of the AST by a *tree-walker* routine that takes the appropriate action at each node.

[*Tree-Walker Functions as Interpreter*]

```
interpretStatementList( statementList, symbolTable )
  for statement in statementList
    switch statement.kind
      case ASSIGNMENT
        symbolTable[ statement.lvalue ].value =
          evaluateExpression( statement.rvalue, symbolTable  )

      case IF
        if evaluateExpression(
            statement.test, symbolTable  ) then
          interpretStatementList(
            statement.thenSide, symbolTable  )

        else
          interpretStatementList(
            statement.elseSide, symbolTable  )

      case WHILE
        while evaluateExpression(
          statement.test, symbolTable ) do
          interpretStatementList(
            statement.body, symbolTable )

      default
        print "WTF?  I don't understand", statement.kind,
          "as a statement kind."
```

```
evaluateExpression( expression, symbolTable )
  switch expression.kind
    case LITERAL
      return expression.value

    case BINARY_OPERATOR
      return performBOP(
        expression.operator,
        evaluateExpression( expression.left, symbolTable ),
        evaluateExpression( expression.right, symbolTable ) )

    case UNARY_OPERATOR
      return performUOP(
        expression.operator,
        evaluateExpression( expression.left, symbolTable ) )

    default
      print "WTF?  I don't understand", expression.kind,
        "as an expression kind."
      return 0
```

Overview of Compilation

- Many compilers use the AST as the *intermediate form (IF)* handed off to the back end for code generation.
- Others compilers *tree walk* the AST and generate a different intermediate form.
  - A common IF is a *control-flow graph*, whose nodes are fragments of assembly language for an idealized machine.

Overview of Compilation

- Target Code Generation
  - Translates the intermediate form into the target language.
  - Generating code that *works* is not all that hard.
  - Generating *good* code is trickier.

```
     pushl   %ebp              # \
     movl    %esp, %ebp        #  ) reserve space for local variables
     subl    $16, %esp         # /
     call    getint            # read
     movl    %eax, -8(%ebp)    # store i
     call    getint            # read
     movl    %eax, -12(%ebp)   # store j
A:   movl    -8(%ebp), %edi    # load i
     movl    -12(%ebp), %ebx   # load j
     cmpl    %ebx, %edi        # compare
     je      D                 # jump if i == j
     movl    -8(%ebp), %edi    # load i
     movl    -12(%ebp), %ebx   # load j
     cmpl    %ebx, %edi        # compare
     jle     B                 # jump if i < j
     movl    -8(%ebp), %edi    # load i
     movl    -12(%ebp), %ebx   # load j
     subl    %ebx, %edi        # i = i - j
     movl    %edi, -8(%ebp)    # store i
     jmp     C
B:   movl    -12(%ebp), %edi   # load j
     movl    -8(%ebp), %ebx    # load i
     subl    %ebx, %edi        # j = j - i
     movl    %edi, -12(%ebp)   # store j
C:   jmp     A
D:   movl    -8(%ebp), %ebx    # load i
     push    %ebx              # push i (pass to putint)
     call    putint            # write
     addl    $4, %esp          # pop i
     leave                     # deallocate space for local variables
     mov     $0, %eax          # exit status for program
     ret                       # return to operating system
```

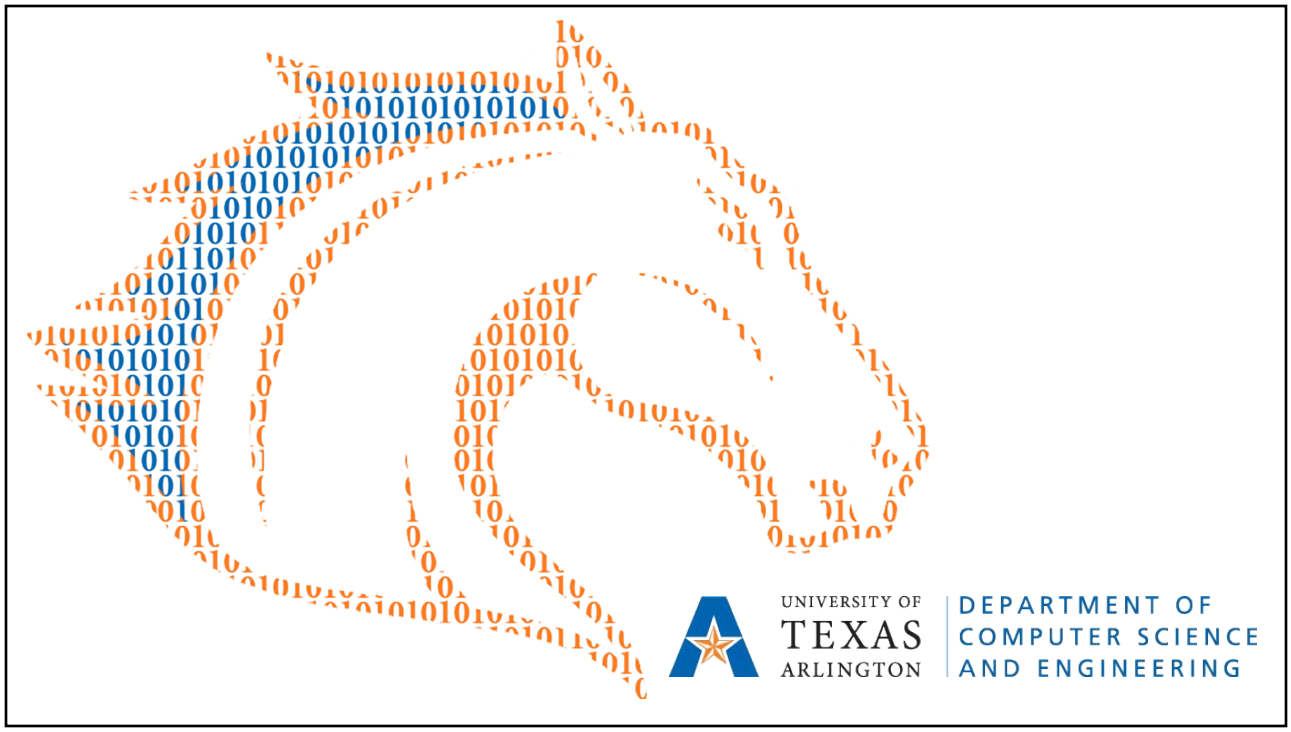
Overview of Compilation

- Code Improvement
  - Often referred to as *optimization*, though that's a bit presumptuous.
  - Not required, but often done in an attempt to improve the generated code.
  - In this case, the optimizer did fairly well.
    - Got rid of most loads and stores as it was able to keep the values in the registers.

```
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    andl  $-16, %esp
    call  getint
    movl  %eax, %ebx
    call  getint
    cmpl  %eax, %ebx
    je    C
A:  cmpl  %eax, %ebx
    jle   D
    subl  %eax, %ebx
B:  cmpl  %eax, %ebx
    jne   A
C:  movl  %ebx, (%esp)
    call  putint
    movl  -4(%ebp), %ebx
    leave
    ret
D:  subl  %ebx, %eax
    jmp   B
```

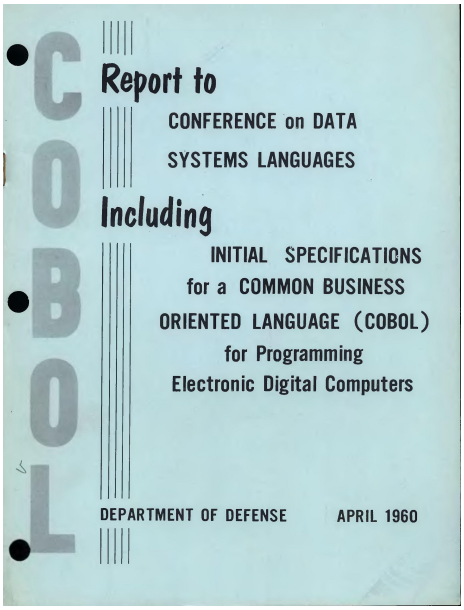
Overview of Compilation

- Code Improvement
  - The previous optimization was at the target language level.
  - Another place code improvement can happen is much earlier in the compilation process, right after semantic analysis.
  - Generally, the earlier an optimization can be made, the greater the effect (improvement, we hope) on the final target program.


UNIVERSITY OF
TEXAS
ARLINGTON
DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Compilers

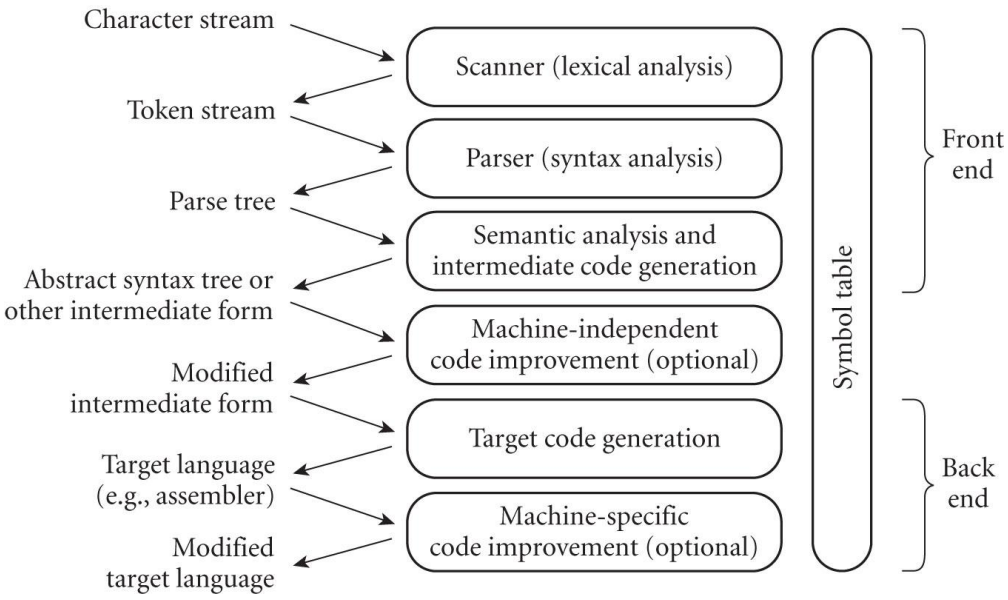
CSE 4305 / CSE 5317
M01 Lexical Analysis
Fall 2020



M01

Lexical Analysis

Phases of Compilation



Program Analysis

- A compiler's *Front End* is concerned with the *Analysis* of the source program.
  - Determining the *Meaning* of the source program.
- Generally thought of as breaking apart into three *phases*:
  - *Lexical Analysis*: Convert a text source program into *tokens*.
  - *Syntactic Analysis*: Convert a token stream into a *parse tree*.
  - *Semantic Analysis*: Enforce semantic rules and convert a parse tree into an *intermediate form*.
    - Two activities, but often happen in an interleaved fashion.

Lexical Analysis

- Take a stream of individual characters and convert it into a stream of *tokens* (possibly with *attributes*).
- Detect *lexical* errors (badly formed literals, illegal characters, etc.)
  - But *not* misspelled keywords. (Why not?)
- Discard whitespace.
  - Useful only inside literals and to separate otherwise ambiguous constructs.
- Discard comments.
  - Some toolsets interpret comments as directives.

Tokens

- A *token* is the basic building block of a program.
  - The shortest strings of characters in the source program that have individual meaning.
- Tokens come in various types, according to the programming language's specification.
  - Common types include *numbers*, *identifiers*, *keywords*, *operators*, *punctuation marks*, and so forth.

Some Generic Token Categories …

Pattern	*Category*	*Attribute*
letter or underscore possibly followed by letters, underscores, or decimal digits	`ID`	symbol table entry
decimal digits	`INTEGER_LITERAL`	int number
" characters "	`STRING_LITERAL`	string
`if`	`IF`	
`<` *or* `<=` *or* `==` *or* `>=` *or* `>`	`RELATIONAL_OPERATOR`	enum value
`[0-9]*(([0-9][.])|([.][0-9]))[0-9]*`	`REAL_LITERAL`	FP number
`(`	`LEFT_PARENTHESIS`	

Example …

```
if ( a <= 17.34 ) {
 b = b + 1;
} else {
 // Oops!
 print( "too big!" );
}
```

Characters	*Token*	*Attribute*
if	IF	
(	LEFT_PARENTHESIS	
a	ID	"a"
<=	RELATIONAL_OPERATOR	LE
17.34	REAL_LITERAL	17.34
)	RIGHT_PARENTHESIS	
{	LEFT_BRACE	
b	ID	"b"
=	ASSIGN_OPERATOR	ASSIGN
b	ID	"b"
+	ADD_OPERATOR	PLUS
1	INTEGER_LITERAL	1
;	SEMICOLON	
}	RIGHT_BRACE	
else	ELSE	
{	LEFT_BRACE	
print	ID	"print"
(	LEFT_PARENTHESIS	
"too big!"	STRING_LITERAL	"too big!"
)	RIGHT_PARENTHESIS	
;	SEMICOLON	
}	RIGHT_BRACE	

Lexical Analysis

- So how to do this conversion?
- We could just hand-code a routine …
- Cool, huh?

```
Tokens for "fred _ 15 1234.345 "bob" Maddog87":
    ID 'fred'
    ID '_'
    INTEGER_LITERAL 15
    INTEGER_LITERAL 1234
Illegal character '.'
    INTEGER_LITERAL 345
Illegal character '"'
    ID 'bob'
Illegal character '"'
    ID 'Maddog87'
```

```
void tokenize( char *inStr ) {
  int ptr = 0;

  while ( inStr[ ptr ] ) {
    if ( isspace( inStr[ ptr ] ) ) {
      ptr += 1;

    } else if ( isdigit( inStr[ ptr ] ) ) {
      int n = 0;

      while ( isdigit( inStr[ ptr ] ) ) {
        n = n*10 + inStr[ ptr ]-'0';
        ptr += 1;
      }

      printf( "    INTEGER_LITERAL %d\n", n );

    } else if ( isalpha( inStr[ ptr ] ) || inStr[ ptr ] == '_' ) {
      int ptrBegin = ptr;

      while ( isalpha( inStr[ ptr ] ) ||
              isdigit( inStr[ ptr ] ) ||
              inStr[ ptr ] == '_' ) {
        ptr += 1;
      }

      printf( "    ID '%.*s'\n", ptr-ptrBegin, &inStr[ptrBegin] );

    } else {
      printf( "Illegal character '%c'\n", inStr[ ptr ] );
      ptr += 1;
    }
  }
```

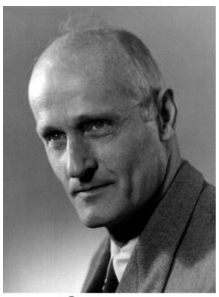
Lexical Analysis

- Hand-coded lexical analyzers are fun for about 10 seconds and then one realizes that they're …
  - Tedious to write.
  - Difficult to extend.
    - Imagine adding `REAL_LITERAL` to this tokenizer.
  - Error-prone.
    - Imagine adding `REAL_LITERAL` to this tokenizer and not screwing up `INTEGER_LITERAL` along the way.
  - Hard to separate into *patterns* and *processing*.

Lexical Analysis

- We want to express the *patterns* of token classes as *Regular Expressions*.
  - Easy to write, easy to update, reduced chance of errors, lots of theory to help with the processing.
- We want to express the *processing* of token classes independently of the *pattern*.
  - Reduced chance of error, easier to write, easier to update.

Regular Expression (RE)



Stephen C. Kleene

/ˈkleɪni/ KLAY-nee

1. A *character*
2. The *empty string*, denoted as ε
3. The *concatenation* of two REs

   Meaning one RE after the other RE
4. The *alternation* of two REs, denoted by |

   Meaning one RE *or* the other RE
5. An RE followed by the *Kleene star*, denoted as *

   Meaning *zero* or *more* repetitions of the RE

https://www.nap.edu/read/9649/chapter/10

Regular Expressions

- Aside from the basic five items, there are some common notation extensions that make writing regular expressions more convenient.
- An RE followed by the *Kleene Plus*, denoted by `+`
  - Meaning *one* or *more* repetitions of the RE.
  - `a+` is equivalent to `aa*`, so no extra power.
- An RE followed by `?`
  - Meaning *zero* or *one* instances of the RE.
  - `a?` is equivalent to `(a|ε)`, so no extra power.
- A set or range of characters in brackets, `[]`
  - Meaning any *one* of the indicated characters.
  - `[0-9]` is equivalent to `0|1|2|3|4|5|6|7|8|9`, so no extra power.

Regular Expressions

- Regular Expressions are just one category of *formal* languages.
- These categories can be organized in a hierarchy according to the kinds of languages they can describe (and their parsing complexity, required resources, etc.)
- Different levels have different rules for their *productions*.

Grammars and the Chomsky Language Hierarchy

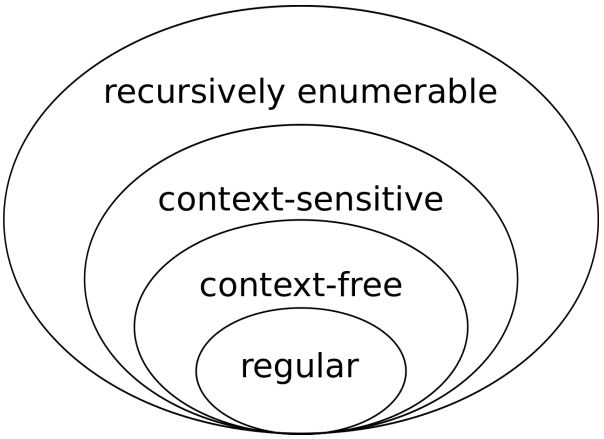
- A *grammar* specification includes
  - A set of *terminal* symbols (T)
  - A set of *non-terminal* symbols (N)
    - $V = T \cup N$
  - A set of *production rules*
  - A *start symbol*

Terminals	a, b
Non-terminals	S
Production rules	$S \rightarrow aSb$ $S \rightarrow \varepsilon$
Start symbol	S

- Restrictions on the form of the production rules can be used to categorize grammars into classes, each of which is more *expressive* than the previous.
- The example expresses the language $a^n b^n$, for $n \geq 0$.

Grammars and the Chomsky Language Hierarchy

- The Chomsky Hierarchy shows four levels of grammar expressivity.
- Each properly includes the previous.
- The levels have increasing expressiveness, but also parsing complexity, resource requirements, etc.



https://commons.wikimedia.org/wiki/File:Chomsky-hierarchy.svg

[*Grammars and the Chomsky Language Hierarchy*]

Type	*Name*	*Allowed Productions*	*Example Language*	*Example Grammar*	*Example Use*	*Recognizing Automaton*	*Storage Required*	*Parsing Complexity*
0	Recursively Enumerable	$\alpha \rightarrow \beta$ $\alpha \in V+$ $\beta \in V^*$			(*Theoretical Interest*)	Turing Machine	Infinite Tape	Undecidable
1	Context Sensitive	$\alpha \rightarrow \beta$ $S \rightarrow \varepsilon$ $\lvert\alpha\rvert \leq \lvert\beta\rvert$ $\alpha \in V^*NV^*$ $\beta \in V+$ *S not on any RHS*	$a^n b^n c^n$ $n > 0$	$S \rightarrow aSBC$ $S \rightarrow aBC$ $CB \rightarrow BC$ $aB \rightarrow ab$ $bB \rightarrow bb$ $bC \rightarrow bc$ $cC \rightarrow cc$	(*Theoretical Interest*)	Linear Bounded Automaton	Tape a linear multiple of input length	NP Complete
2	Context Free	$A \rightarrow \alpha$ $A \in N$ $\alpha \in V^*$	$a^n b^n$ $n \geq 0$	$S \rightarrow aSb$ $S \rightarrow \varepsilon$	Arithmetic Expressions, Programming Languages	Pushdown Automaton	Pushdown stack	$O(n^3)$
3	Regular	$A \rightarrow xB$ $A \rightarrow x$ $A, B \in N$ $x \in T^*$	$a^n b$ $n > 0$	$S \rightarrow aS$ $S \rightarrow ab$	Token Formats, String Recognition	Finite Automaton	Finite	$O(n)$

[*"Recursive" Production Rules …*]

- Later we'll learn that *recursion* is what separates *Context-Free Grammars* from *Regular Expressions*. No recursion in REs.
- On the previous chart, $S \rightarrow aS$ appears as a rule for an RE.
  - Hey, isn't that *recursive*? After all, *S* refers to itself, right?

- ***No!*** It just ***looks*** as if it's recursive. :)
- Because the *S* appears on the far right (there's nothing after *S* in the rule), this is just a way of expressing a Kleene-* operation (a*).
- A proof that this isn't true recursion requires going deeper in formal language theory than required for this class, so just accept it as true.

Lexical Analysis

Token Class	*Regular Expression*
ID	[_a-zA-Z][_a-zA-Z0-9]*
INTEGER_LITERAL	[0-9]+
STRING_LITERAL	"[^"\n]*"
IF	if
RELATIONAL_OPERATOR	<|<=|==|>=|>
REAL_LITERAL	[0-9]*(([0-9][.])|([.][0-9]))[0-9]*
LEFT_PARENTHESIS	(

Processing Regular Expressions

- How to use Regular Expressions to do lexical scanning?
  - That is, how do we convert an RE into a program?
- Recognize that there's a correspondence between an RE and a *Finite Automaton*.
  - And a *Finite Automaton* is convertible into *scanning* (or *recognizing*) program in a mechanical way.

Finite Automaton (FA)

- A *Finite Automaton* consists of …
  - A finite set of *symbols*, known as its *alphabet*.
  - A finite set of *states*.
  - A finite set of *edges* each of which …
    - … goes from one state to another (possibly the same) state.
    - … is labelled with a *symbol*.
  - One identified state known as the *start state*.
    - Usually state 1.
  - A subset of states identified as *final* (or *accepting*) states.

Finite Automaton (FA)

- A finite automaton *scans* a *finite* input string by …
  - Starting in the *start* state.
  - For each successive *symbol* in the input string, transiting along an edge from the current state that is labelled with the symbol.
  - *Or* at any time transiting along an accessible edge labelled with ε.

- After all symbols are used up (and all ε moves are made), if the current state is a *final* state, the FA *accepts* the input, otherwise it *rejects* the input.

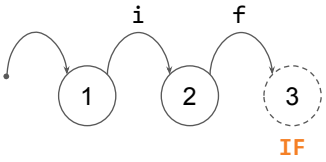
Finite Automaton (FA)

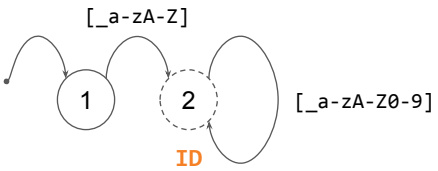
- The set of strings *accepted* by an FA is said to be its *language*.
- Though all FA are *finite*, the *language* accepted by an FA can be *infinite*.
  - Each *string* in the language, however, is itself *finite*.

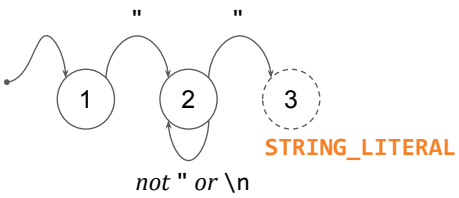
Deterministic Finite Automaton (DFA)

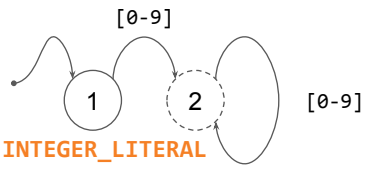
- If a Finite Automaton complies with the following two criteria, it is said to be *Deterministic* …
  - No two edges from a state can have the *same* symbol.
    - Each symbol valid for a state appears on only *one* edge.
  - No edge is labelled with ε.
    - ε is the *empty* transition, i.e., no symbol is consumed.
- When a FA is *deterministic*, the acceptance or rejection of an input string will take no more transitions than the length of the input.
  - Why?

Deterministic Finite Automaton Examples



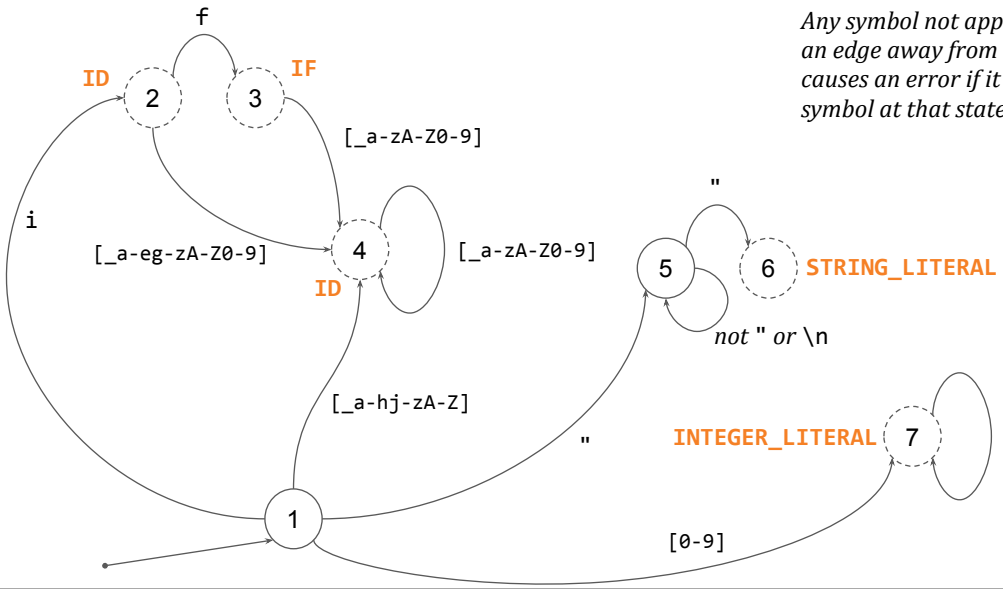






Any symbol not appearing in an edge away from a state causes an error if it is the next symbol at that state.

Combining Deterministic Finite Automata



Any symbol not appearing in an edge away from a state causes an error if it is the next symbol at that state.

Lexical Analysis

- What a pain in the butt!
- Imagine having to do that kind of construction for a programming language with dozens of overlapping token class definitions.
- However, we can express this complex Finite Automaton in a more convenient form …

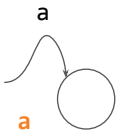
Nondeterministic Finite Automata (NFA)

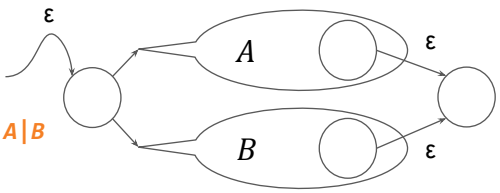
- If an FA satisfies either of the two following criteria, it is *Non-Deterministic* …
  - At least one state has at least two edges going to different other states and bearing the the same symbol.
  - At least one edge is labelled with ε.
- Does being *non-deterministic* increase the descriptive power of a Finite Automaton?
  - ***No!*** Why not?

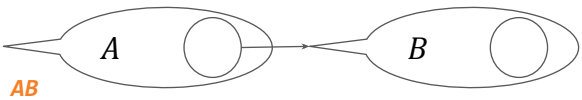
Making a Regular Expression into an NFA

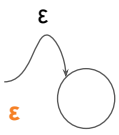
- Turns out that it's trivial to convert a Regular Expression into an NFA.
- We'll show this *Constructively* by giving ways to convert each of the five kinds of Regular Expressions into an NFA.
- In the following, a is any symbol, ε is the empty string, and *A* and *B* are any NFAs representing regular expressions.

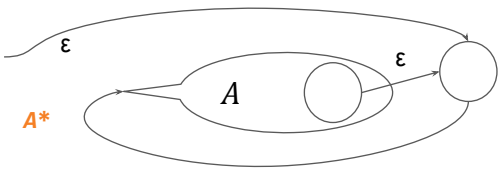
Making an RE into an NFA



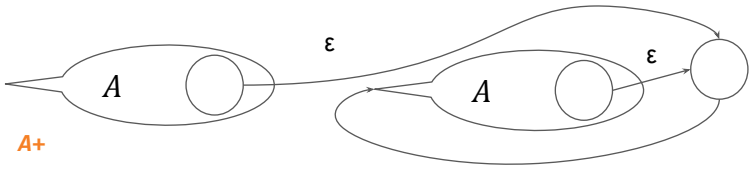




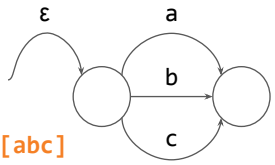




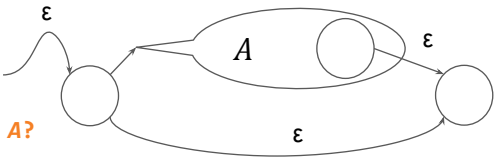
Making an RE into an NFA



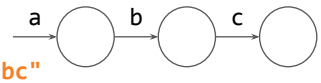
A+



[abc]

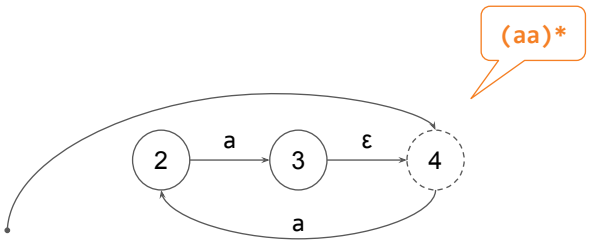


A?

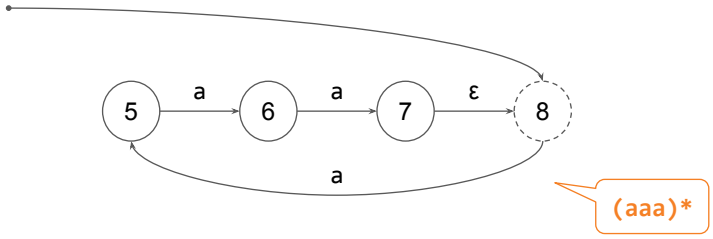


"abc"

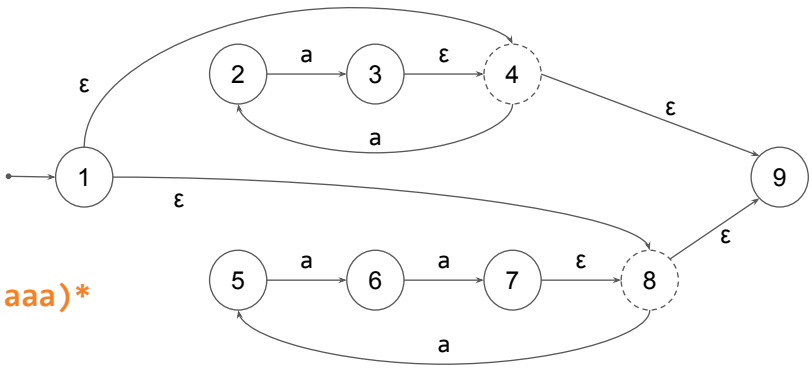
Making an RE into an NFA Example …



Making an RE into an NFA Example …



Making an RE into an NFA Example …



(aa)*|(aaa)*

Using an NFA for Lexical Scanning

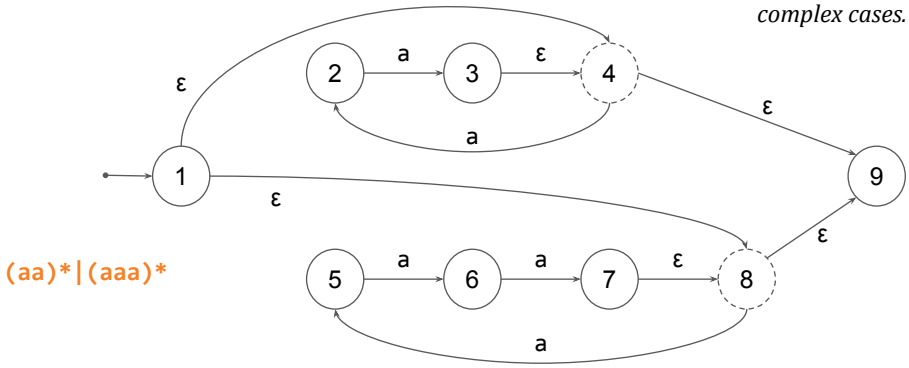
- It was trivial to convert that RE into an NFA.
- What about using the NFA for lexical scanning?
- This is not so trivial.
- As symbols are consumed, we might have to *guess* which edge to traverse.
  - If a state has multiple outgoing edges with the same symbol.
- We also might have to *guess* whether to use an edge or not.
  - If a state has outgoing edge(s) with ε.
- FYI: Few computers have good guessing hardware.

Converting an NFA into a DFA

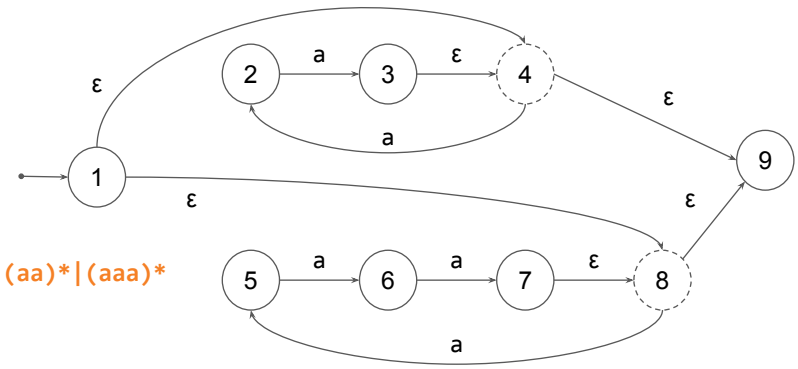
- We have to keep track of whichever state the NFA *might* be in at any point.
- We do this by taking the current state and computing its ε-*closure*.
  - That is, all states that can be reached from it solely by ε transitions (perhaps several in a row …).
- This results in a (possibly) *combined state*.
- For that combined state, we compute the ε-closure of the destination combined states for each symbol on outgoing edges.
- And then repeat until we formed the total set of possible ε-closed combined states …

NFA to DFA Example …

This is a simple example with only one symbol, a. *The process is the same for more complex cases.*

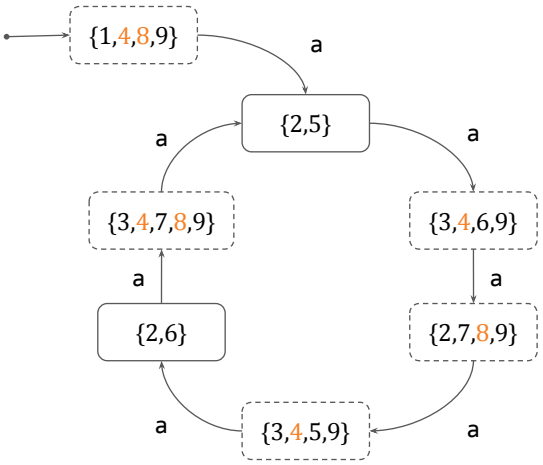


NFA to DFA Example …



State	a⟶	ε⟶
Start {1}	—	{1,4,8,9}
{1,4,8,9}	{2,5}	{2,5}
{2,5}	{3,6}	{3,4,6,9}
{3,4,6,9}	{2,7}	{2,7,8,9}
{2,7,8,9}	{3,5}	{3,4,5,9}
{3,4,5,9}	{2,6}	{2,6}
{2,6}	{3,7}	{3,4,7,8,9}
{3,4,7,8,9}	{2,5}	{2,5}

NFA to DFA Example …



State	a⟶	ε⟶
Start {1}	—	{1,4,8,9}
{1,4,8,9}	{2,5}	{2,5}
{2,5}	{3,6}	{3,4,6,9}
{3,4,6,9}	{2,7}	{2,7,8,9}
{2,7,8,9}	{3,5}	{3,4,5,9}
{3,4,5,9}	{2,6}	{2,6}
{2,6}	{3,7}	{3,4,7,8,9}
{3,4,7,8,9}	{2,5}	{2,5}

(aa)*|(aaa)*

Converting an NFA into a DFA

- Once the complete set of combined states is generated, the set of *final states* can be identified.
- A combined state that includes *any* final state from the original NFA is a final state in the DFA.
  - These are marked with a special color in the previous slides.
  - The final combined states are *dashed* boxes.

Converting a DFA to a Program

- As we stated previously, deciding if a DFA *accepts* a string is O(n), where n is the length of the string.
  - Every transition of the DFA is deterministic and therefore consumes a symbol from the input string.
- It's not hard to write a program that mechanically converts a set of *Regular Expressions* (and their corresponding action routines) into a *program*.
  - Such a program is a *Lexical Analyzer Generator*.

Lexical-Analyzer Generators

- Remember that we said an advantage of REs as a formalism is that there's lots of theory already developed to help us.
- `lex` was developed in 1975 to take sets of regular expressions and actions associated with each and automatically generate a lexical scanner.
  - Worked with C under UNIX.
- Zillions of subsequent tools were created to do the same kind of thing but with different notations, different target languages, different environments, etc.

`flex`

- We will be using `flex` (***f***ast ***lex***ical analyzer generator) …
  - A derivative of the original `lex` tool.

```
pi@icywits-01:~ $ apt-cache policy flex
flex:
  Installed: 2.6.4-6.2
  Candidate: 2.6.4-6.2
  Version table:
 *** 2.6.4-6.2 500
        500 http://raspbian.raspberrypi.org/raspbian buster/main armhf Packages
        100 /var/lib/dpkg/status
pi@icywits-01:~ $ flex --version
flex 2.6.4
pi@icywits-01:~ $
```

Lexical Analysis

- Remember the hand-coded tokenizer routine?
- We thought it was pretty cool, huh?

```
Tokens for "fred _ 15 1234.345 "bob" Maddog87":
    ID 'fred'
    ID '_'
    INTEGER_LITERAL 15
    INTEGER_LITERAL 1234
Illegal character '.'
    INTEGER_LITERAL 345
Illegal character '"'
    ID 'bob'
Illegal character '"'
    ID 'Maddog87'
```

```
void tokenize( char *inStr ) {
  int ptr = 0;

  while ( inStr[ ptr ] ) {
    if ( isspace( inStr[ ptr ] ) ) {
      ptr += 1;

    } else if ( isdigit( inStr[ ptr ] ) ) {
      int n = 0;

      while ( isdigit( inStr[ ptr ] ) ) {
        n = n*10 + inStr[ ptr ]-'0';
        ptr += 1;
      }

      printf( "    INTEGER_LITERAL %d\n", n );

    } else if ( isalpha( inStr[ ptr ] ) || inStr[ ptr ] == '_' ) {
      int ptrBegin = ptr;

      while ( isalpha( inStr[ ptr ] ) ||
              isdigit( inStr[ ptr ] ) ||
              inStr[ ptr ] == '_' ) {
        ptr += 1;
      }

      printf( "    ID '%.*s'\n", ptr-ptrBegin, &inStr[ptrBegin] );

    } else {
      printf( "Illegal character '%c'\n", inStr[ ptr ] );
      ptr += 1;
    }
  }
```

`flex` Version

- Each token type has its own processing routine.
- Token formats expressed as regular expressions.
- Cool, huh?

```
Tokens for "fred _ 15 1234.345 "bob" Maddog87":
    ID 'fred'
    ID '_'
    INTEGER_LITERAL 15
    INTEGER_LITERAL 1234
Illegal character '.'
    INTEGER_LITERAL 345
Illegal character '"'
    ID 'bob'
Illegal character '"'
    ID 'Maddog87'
```

```
[0-9]+  {
  printf( "    INTEGER_LITERAL %s\n", yytext );
}

[_a-zA-Z][_a-zA-Z0-9]*  {
  printf( "    ID '%s'\n", yytext );
}

[ \t\r\n] { /* Ignore whitespace */ }

.       { printf( "Illegal character '%s'\n", yytext ); }
```

`flex` Version (2)

- Now with `REAL_LITERAL`.
- Not hard to add.
- *Really* cool, huh?

```
Tokens for "fred _ 15 1234.345 "bob" Maddog87":
    ID 'fred'
    ID '_'
    INTEGER_LITERAL 15
    REAL_LITERAL 1234.345
Illegal character '"'
    ID 'bob'
Illegal character '"'
    ID 'Maddog87'
```

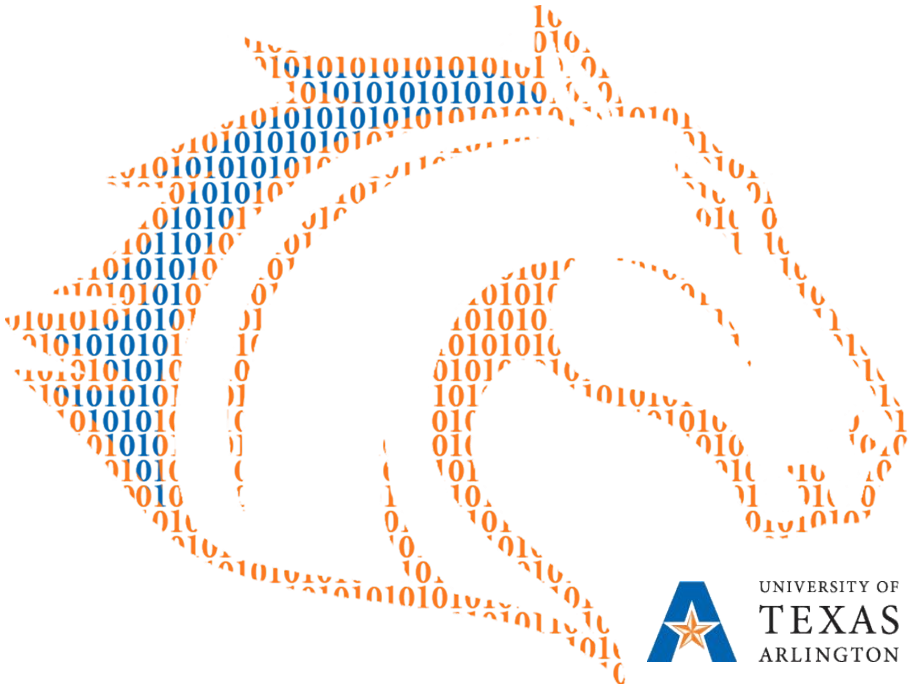
```
[0-9]+  {
  printf( "    INTEGER_LITERAL %s\n", yytext );
}

[0-9]+\.[0-9]+ {
  printf( "    REAL_LITERAL %s\n", yytext );
}

[_a-zA-Z][_a-zA-Z0-9]*  {
  printf( "    ID '%s'\n", yytext );
}

[ \t\r\n] { /* Ignore whitespace */ }

.       { printf( "Illegal character '%s'\n", yytext ); }
```

`flex` Version (3)

- Now with `STRING_LITERAL`.
- Again, not hard to add.
- *Really, really* cool, huh?

```
Tokens for "fred _ 15 1234.345 "bob" Maddog87":
    ID 'fred'
    ID '_'
    INTEGER_LITERAL 15
    REAL_LITERAL 1234.345
    STRING_LITERAL "bob"
    ID 'Maddog87'
```

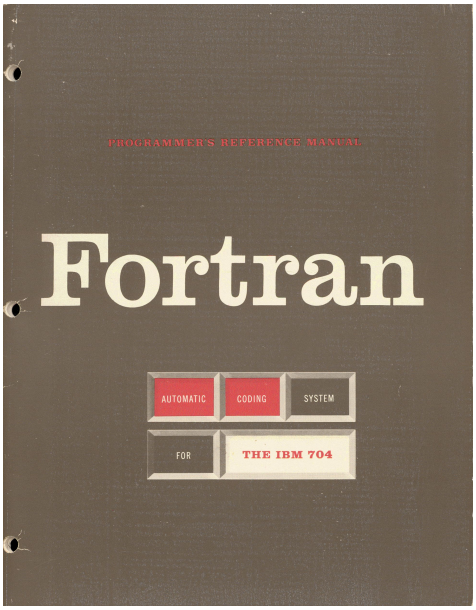
```
[0-9]+  {
  printf( "    INTEGER_LITERAL %s\n", yytext );
}

[0-9]+\.[0-9]+  {
  printf( "    REAL_LITERAL %s\n", yytext );
}

["][^"\n]*["]  {
  printf( "    STRING_LITERAL %s\n", yytext );
}

[_a-zA-Z][_a-zA-Z0-9]*  {
  printf( "    ID '%s'\n", yytext );
}

[ \t\r\n]  { /* Ignore whitespace */ }

.        { printf( "Illegal character '%s'\n", yytext ); }
```



UNIVERSITY OF TEXAS ARLINGTON | DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Compilers

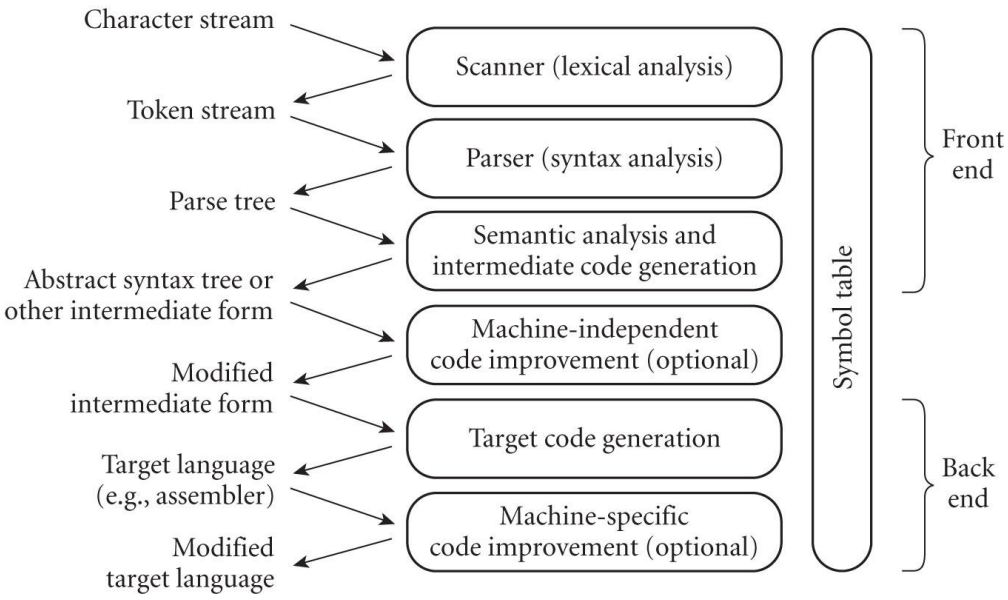
CSE 4305 / CSE 5317
M02 Syntactic Analysis
Fall 2020



M02

Syntactic Analysis

Phases of Compilation



Program Analysis

- A compiler's *Front End* is concerned with the *Analysis* of the source program.
  - Determining the *Meaning* of the source program.
- Generally thought of as breaking apart into three *phases*:
  - *Lexical Analysis*: Convert a text source program into *tokens*.
  - *Syntactic Analysis*: Convert a token stream into a *parse tree*.
  - *Semantic Analysis*: Enforce semantic rules and convert a parse tree into an *intermediate form*.
    - Two activities, but often happen in an interleaved fashion.

Syntactic Analysis

- Take a stream of *tokens* and convert it into a *parse tree*.
- Capture the *hierarchical* structure of the program.
  - Declarations, definitions, blocks, statements, expressions, …
- Provide representation for subsequent *semantic* analysis.
- Detect *syntactic* errors.
  - Mostly, improper structure, including misspelled keywords, bad statement and expression construction.
  - But *not*, e.g., undeclared identifiers, mismatched function calls.
  - (Why not?)

Syntactic Analysis

- So how to do this analysis?
- In the *Lexical Analysis* phase, we used a formal specification of the token formats.
  - Regular expressions (with action routines).
- Can we use Regular Expressions as the notation for the formal specification of program structure?
- *No!*

Why Not Regular Expressions?

- We have shown that Regular Expressions can be directly converted to an NFA, which can then be converted to a DFA.
- And that DFA can then be used to accept or reject an input string as belonging or not belonging to the RE's language.

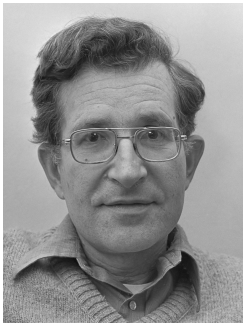
- The important letter here is ***F***, for *finite*.
- That means that the DFA cannot be used to recognize, e.g., *nesting* that is "too deep".
  - There just won't be enough states in the DFA.

Context-Free Grammar (CFG)

- Regular Expressions are limited in expressiveness.
  - Cannot define strings that are required to have …
    - Well-formed and/or nested parentheses, brackets, etc.
      e.g., `([[{(){}[][]}]{[]}])`
    - Matching pairs
      e.g., $a^n b^n \longrightarrow$ ab, aabb, aaabbb, …

- Next step up in expressiveness is a *Context-Free Grammar*.
  - Definitions can refer to themselves, i.e., can *recurse*.

Context-Free Grammar



Noam Chomsky

1. A set of *Terminal Symbols*.

   The *tokens* generated by the lexical analyzer.

2. A set of *Non-terminal Symbols*.

   Representing syntactic categories.
   Distinguished from the *Terminal Symbols*.

3. A set of *Production Rules*.

   Relating the syntactic categories to their structure. The LHS of each must be a single *Non-Terminal Symbol*. The RHS of each can be arbitrarily complex.

4. A *Start Symbol*.

   One of the *Non-terminal Symbols*.

https://commons.wikimedia.org/wiki/File:Noam_Chomsky_(1977).jpg

CFG Example

expr ⟶ *id* | *number* | - *expr* | (*expr*) | *expr op expr*

id ⟶ (_ | a | b | … | z)(_ | a | b | … | z | 0 | 1 | … | 9)*

op ⟶ + | - | * | /

- Notice that *expr* refers to itself. This definition is *recursive*.
  - It's *not* left or right recursive exclusively.
- Q: What set of strings does *id* define?

CFG Example

- To *derive* (or *generate*) a string from a CFG, begin with the start symbol and replace non-terminals according to the rules until only terminals remain.
  - A *sentential form* is the start symbol or any form derived from it.
  - A *sentence* is a sentential form which has only terminal symbols.
- Example, generate

  `slope * x + intercept`

  using the *expr* CFG.

expr

expr op expr

expr op id

expr + *id*

expr op expr + *id*

expr op id + *id*

expr * *id* + *id*

id * *id* + *id*

`slope * x + intercept`

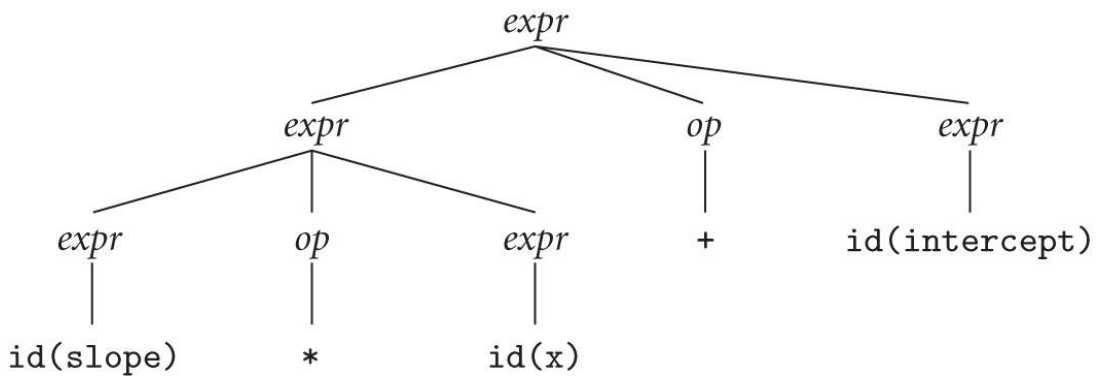
CFG Example

- That derivation was *Right-Most*.
  - We replaced the *right-most* non-terminal each time we took a step in the derivation.
  - *Except* for the final replacement of the *id* non-terminals with their actual words. (This was to make the derivation a little shorter and easier to fit on the slide.)
- A *Left-Most* derivation replaces the *left-most* non-terminal each time a step is taken in the derivation. (Duh.)

CFG Example

- The steps we took in the derivation correspond to the construction of a *parse tree*.
- The *root* of the parse tree is the *start symbol*.
- Every time we use a production rule, it's the same as adding a new (set of) node(s) to the tree.
- All internal nodes of the parse tree are *non-terminals*.
- The leaves of the final parse tree are *terminals*.
  - These terminals are the *tokens* of the original string.

CFG Example Parse Tree (from the Right)



CFG Example

- There's another way to do the generation of

  `slope * x + intercept`

  using the *expr* CFG.
- This derivation goes from the *left* instead of the *right*.

expr

expr op expr

id op expr

*id * expr*

*id * expr op expr*

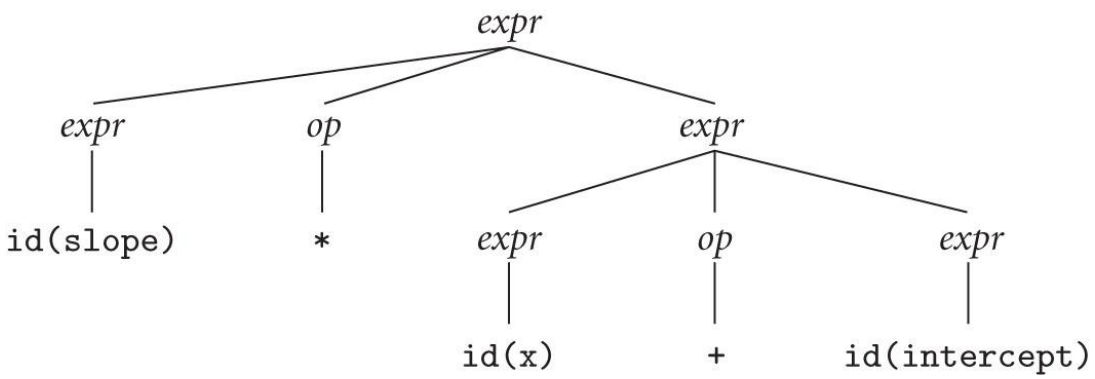
*id * id op expr*

*id * id + expr*

*id * id + id*

`slope * x + intercept`

CFG Example Parse Tree (from the Left)



CFG Example

- The CFG allows two parse trees for the same string because its definition is *ambiguous*.
  - Also, it allows incorrect parsing of operator precedence.
- So is the answer to always derive from the right?
  - After all, that got the correct derivation.
- ***No!***
  - Consider the right-most derivation of `intercept + slope * x`

expr

expr op expr

expr op id

expr * *id*

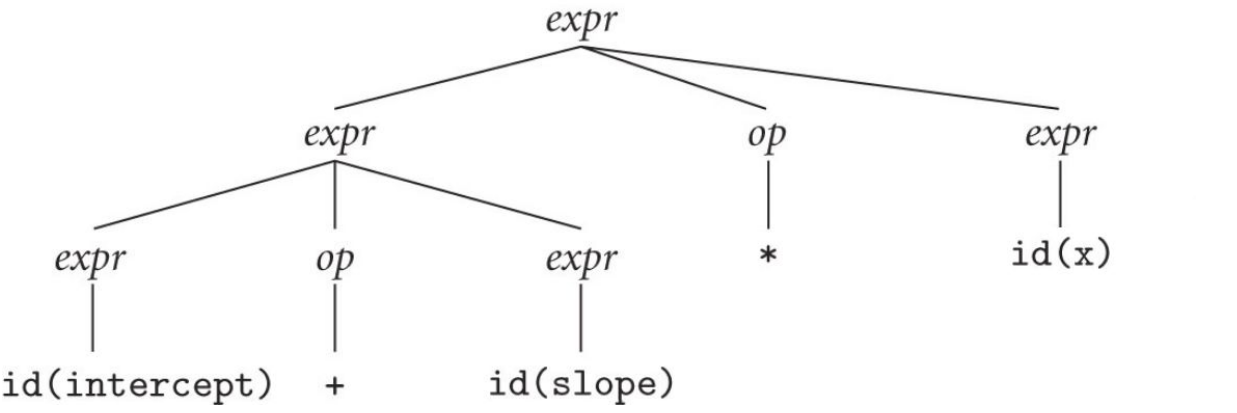
expr op expr * *id*

expr op id * *id*

expr + *id* * *id*

id + *id* * *id*

`intercept + slope * x`



CFG Example 2 Parse Tree (from the Right)

Improved CFG Example

- We need a CFG that is not ambiguous *and* honors our concept of operator precedence.
- A better (though more complex) CFG would be

expr ⟶ *term* | *expr add_op term*

term ⟶ *factor* | *term mul_op factor*

factor ⟶ *id* | *number* | - *factor* | (*expr*)

add_op ⟶ + | -

mul_op ⟶ * | /

expr

expr add_op term

expr add_op factor

expr add_op id

expr + *id*

term + *id*

term mul_op factor + *id*

term mul_op id + *id*

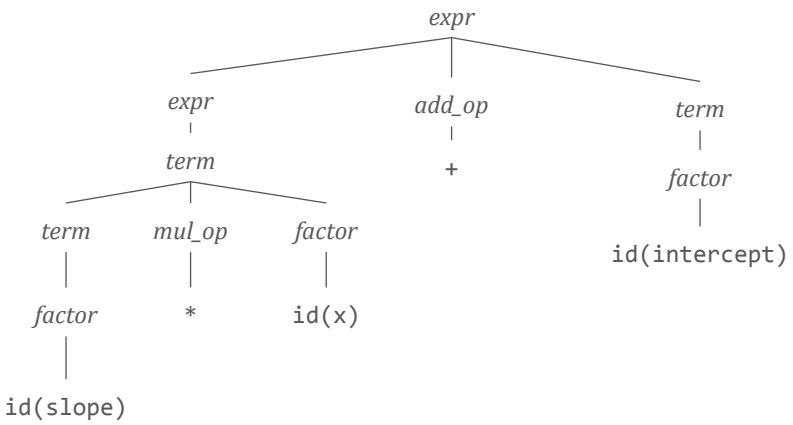
term * *id* + *id*

factor * *id* + *id*

id * *id* + *id*

`slope * x + intercept`

Improved CFG Parse Tree (from the Right)



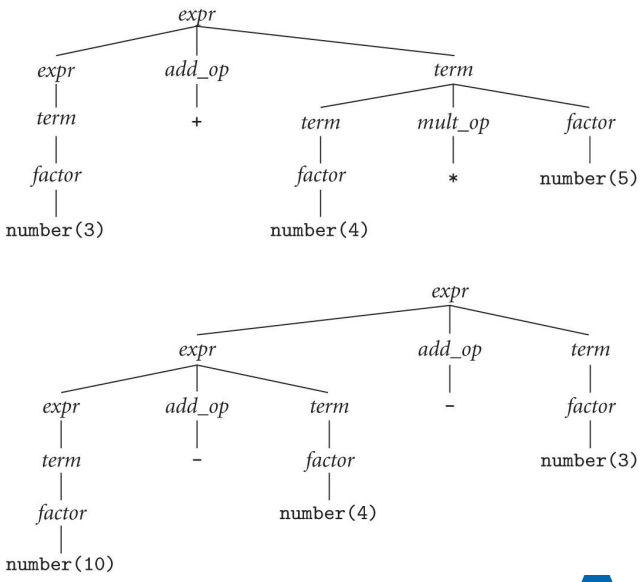
Improved CFG Example

- What about doing the derivation from the left?
- We still get the correct derivation!
  - The grammar is unambiguous and it honors our intuitive understanding of operator precedence.
  - Multiply and divide are "tighter" than addition and subtraction.
- But what about *associativity*?
  - How are "like" operations grouped?

expr
expr add_op term
term add_op term
term mul_op factor add_op term
factor mul_op factor add_op term
id mul_op factor add_op term
*id * factor add_op term*
*id * id add_op term*
*id * id + term*
*id * id + factor*
*id * id + id*
`slope * x + intercept`

Improved CFG Example

- This CFG handles *precedence* and *associativity* according to our intuitive expectations.
  - 3+4*5 means
    - 3+(4*5) = 23
    - *not* (3+4)*5 = 35.
  - 10-4-3 means
    - (10-4)-3 = 3
    - *not* 10-(4-3) = 9.



Improved CFG Example

- OK, so it works. But *why* does it work?
- Operator *precedence* is enforced by having nested rules that permit the repetition of only certain operators at each precedence level.
  - *expr* is "looser" than *term* as it occurs higher in the CFG definition.
  - Therefore the *add_op* operators are "looser" than the *mul_op* operators.
- Operator *associativity* is enforced by having the *expr* and *term* rules recurse on their left side rather than the right.
  - Therefore operations group from the left instead of the right.

expr ⟶ *term* | *expr add_op term*

term ⟶ *factor* | *term mul_op factor*

factor ⟶ `id` | `number` | `-` *factor* | `(` *expr* `)`

add_op ⟶ `+` | `-`

mul_op ⟶ `*` | `/`

Improved CFG Example Comments …

- We "fixed" the grammar's ambiguity by introducing additional productions to keep the operators separate.
- Suppose we want to add *another* operator at yet *another* level of precedence?
  - … and *another* and *another* and … ?
- How far might this have to go? How many levels?

Precedence

- Lots and lots of operators.
- Table goes from highest *precedence* down to lowest.
- Languages try to organize precedence to ease expression writing.

Fortran	Pascal	C	Ada
		`++`, `--` (post-inc., dec.)	
`**`	`not`	`++`, `--` (pre-inc., dec.), `+`, `-` (unary), `&`, `*` (address, contents of), `!`, `~` (logical, bit-wise not)	`abs` (absolute value), `not`, `**`
`*`, `/`	`*`, `/`, `div`, `mod`, `and`	`*` (binary), `/`, `%` (modulo division)	`*`, `/`, `mod`, `rem`
`+`, `-` (unary and binary)	`+`, `-` (unary and binary), `or`	`+`, `-` (binary)	`+`, `-` (unary)
		`<<`, `>>` (left and right bit shift)	`+`, `-` (binary), `&` (concatenation)
`.eq.`, `.ne.`, `.lt.`, `.le.`, `.gt.`, `.ge.` (comparisons)	`<`, `<=`, `>`, `>=`, `=`, `<>`, `IN`	`<`, `<=`, `>`, `>=` (inequality tests)	`=`, `/=`, `<`, `<=`, `>`, `>=`
`.not.`		`==`, `!=` (equality tests)	
		`&` (bit-wise and)	
		`^` (bit-wise exclusive or)	
		`|` (bit-wise inclusive or)	
`.and.`		`&&` (logical and)	`and`, `or`, `xor` (logical operators)
`.or.`		`||` (logical or)	
`.eqv.`, `.neqv.` (logical comparisons)		`?:` (if…then…else)	
		`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `^=`, `|=` (assignment)	
		`,` (sequencing)	

Precedence

- C's ***15*** levels of operator precedence are tricky to remember.
  - But once one knows and uses them effectively, one hardly ever needs to use parentheses for grouping.
  - Watch out! The shift operators (`<<`, `>>`) are lower precedence than the add and subtract (`+`, `-`) operators! `1 << 3 + 1 << 5` is `(1 << (3+1)) << 5`.
- Ada has only ***6*** levels, but the `and` and `or` operators are at the same level of precedence.
  - Hard to believe, but `A or B and C` means `( A or B ) and C`.
- Pascal's ***4*** levels have `and` at a higher precedence than the comparisons.
  - `A < B and C < D` means `A < ( B and C ) < D`, a static error.

WTF?

- Wow.
- Imagine having to make up ***15*** levels of production rules just to get C's operator precedence to work correctly.
  - And then start worrying about *associativity*.

- There has *got* to be a better way to do this!
- Also, how do we get from those *Production Rules* to an actual program?
  - That is, how do we get from a *grammar* to a *parser*?

[*C's Precedence Levels in the Grammar*]

```
primary_expression
  : IDENTIFIER
  | I_CONSTANT | F_CONSTANT | ENUMERATION_CONSTANT
  | STRING_LITERAL | FUNC_NAME
  | '(' expression ')'
  | generic_selection ;

generic_selection
  : GENERIC '(' assignment_expression ',' generic_assoc_list ')' ;

generic_assoc_list
  : generic_association
  | generic_assoc_list ',' generic_association ;

generic_association
  : type_name ':' assignment_expression
  | DEFAULT ':' assignment_expression ;

postfix_expression
  : primary_expression
  | postfix_expression '[' expression ']'
  | postfix_expression '(' ')'
  | postfix_expression '(' argument_expression_list ')'
  | postfix_expression '.' IDENTIFIER
  | postfix_expression PTR_OP IDENTIFIER
  | postfix_expression INC_OP
  | postfix_expression DEC_OP
  | '(' type_name ')' '{' initializer_list '}'
  | '(' type_name ')' '{' initializer_list ',' '}' ;

argument_expression_list
  : assignment_expression
  | argument_expression_list ',' assignment_expression ;

unary_expression
  : postfix_expression
  | INC_OP unary_expression
  | DEC_OP unary_expression
  | unary_operator cast_expression
  | SIZEOF unary_expression
  | SIZEOF '(' type_name ')'
  | ALIGNOF '(' type_name ')' ;
```

```
unary_operator
  : '&' | '*' | '+' | '-' | '~' | '!' ;

cast_expression
  : unary_expression
  | '(' type_name ')' cast_expression ;

multiplicative_expression
  : cast_expression
  | multiplicative_expression '*' cast_expression
  | multiplicative_expression '/' cast_expression
  | multiplicative_expression '%' cast_expression ;

additive_expression
  : multiplicative_expression
  | additive_expression '+' multiplicative_expression
  | additive_expression '-' multiplicative_expression ;

shift_expression
  : additive_expression
  | shift_expression LEFT_OP additive_expression
  | shift_expression RIGHT_OP additive_expression ;

relational_expression
  : shift_expression
  | relational_expression '<' shift_expression
  | relational_expression '>' shift_expression
  | relational_expression LE_OP shift_expression
  | relational_expression GE_OP shift_expression ;

equality_expression
  : relational_expression
  | equality_expression EQ_OP relational_expression
  | equality_expression NE_OP relational_expression ;

and_expression
  : equality_expression
  | and_expression '&' equality_expression ;
```

```
exclusive_or_expression
  : and_expression
  | exclusive_or_expression '^' and_expression ;

inclusive_or_expression
  : exclusive_or_expression
  | inclusive_or_expression '|' exclusive_or_expression ;

logical_and_expression
  : inclusive_or_expression
  | logical_and_expression AND_OP inclusive_or_expression ;

logical_or_expression
  : logical_and_expression
  | logical_or_expression OR_OP logical_and_expression ;

conditional_expression
  : logical_or_expression
  | logical_or_expression '?' expression ':' conditional_expression ;

assignment_expression
  : conditional_expression
  | unary_expression assignment_operator assignment_expression ;

assignment_operator
  : '=' | MUL_ASSIGN | DIV_ASSIGN | MOD_ASSIGN | ADD_ASSIGN | SUB_ASSIGN
  | LEFT_ASSIGN | RIGHT_ASSIGN | AND_ASSIGN | XOR_ASSIGN | OR_ASSIGN ;

expression
  : assignment_expression
  | expression ',' assignment_expression ;
```

Excerpted from http://www.quut.com/c/ANSI-C-grammar-y.html

Some Context-Free Grammar Theory

- As CFGs are a *formal notation* (as are the *regular expressions*), what can we do with them *mechanically*?
  - That is, *automatically* and *without human intelligence*.
- This is important as it helps us understand for which operations we can construct *tools* that will process a CFG in a useful way.
  - Like getting a *parser* from a *grammar*.

Some Context-Free Grammar Theory

- Unfortunately, many useful properties of CFGs are *undecidable*.
  - ❌ Does a CFG generate *all* possible strings of its *terminals*?
  - ❌ Do two CFGs generate the *same* language?
  - ❌ Does a CFG generate *all* strings another CFG generates?
  - ❌ Does a CFG generate *any* string another CFG generates?
  - ❌ Is the *language* a CFG generates *regular*?
  - ❌ Is a CFG *ambiguous*?
- [Take a *theory* class to understand the *why* of these.]

Context-Free Grammar Theory

- On the other hand, some incredibly useful properties *are* decidable.
  - ✅ Is the language a CFG generates *empty*?
  - ✅ Is the language a CFG generates *finite*?
  - ✅ Is a *CFG* regular? (Either *left* or *right*.)
    - ***Not*** "Is the CFG's *language* regular?" as that's *undecidable*.
  - ✅ Does a CFG *generate* a given string?
    - That is, can we *parse* a given string using a given CFG?
  - ✅ Is a non-terminal of a CFG *reachable*, *productive*, or *nullable*?
  - ✅ Is a CFG `LL(1)`?

Predictive Parsing

- So what's that last item, `LL(1)`, mean?
- Essentially, can we *automatically* construct a parser from the CFG that will *efficiently* create an *unambiguous* parse tree from *any* given string?
  - Or tell us the string is *unparseable*.
- `LL(1)` ≡ ***L****eft-to-right* scan of the input, producing a ***L****eftmost* derivation, using only ***1*** token of lookahead.
- This sort of parser never has to *backtrack*.
  - It can parse in time *linear* in the size of the input string.

Predictive Parsing Example

- Let's go back to our improved CFG for expressions.
- It's *unambiguous* already.
- [*id* and *number* will be token categories returned by the scanner.]

expr ⟶ *term* | *expr add_op term*
term ⟶ *factor* | *term mul_op factor*
factor ⟶ *id* | *number* | - *factor* | (*expr*)
add_op ⟶ + | -
mul_op ⟶ * | /

Predictive Parsing Example

- Eliminate explicit alternation to make the production rules more obvious.
- Just make a rule for each case of alternation.

expr ⟶ *expr add_op term*
expr ⟶ *term*
term ⟶ *term mul_op factor*
term ⟶ *factor*
factor ⟶ *id*
factor ⟶ *number*
factor ⟶ - *factor*
factor ⟶ (*expr*)
add_op ⟶ +
add_op ⟶ -
mul_op ⟶ *
mul_op ⟶ /

Predictive Parsing Example

- The *expr* and *term* productions exhibit *left recursion*.
- While the grammar is not ambiguous, it's *non-deterministic* without arbitrary lookahead.
  - `(1+2+3)` vs. `(1+2+3)+4`
  - *expr* ⟶ *term* in the first case but *expr* ⟶ *expr add_op term* in the second.

expr ⟶ *expr add_op term*
expr ⟶ *term*

term ⟶ *term mul_op factor*
term ⟶ *factor*

factor ⟶ *id*
factor ⟶ *number*
factor ⟶ - *factor*
factor ⟶ (*expr*)
add_op ⟶ +
add_op ⟶ -
mul_op ⟶ *
mul_op ⟶ /

Predictive Parsing Example

- *Predictive Parsing* requires that we can *predict* from the next token† which production rule to use.
- We will have to eliminate the left recursion.
  - This can be done mechanically.

expr ⟶ *expr add_op term*
expr ⟶ *term*

term ⟶ *term mul_op factor*
term ⟶ *factor*

factor ⟶ *id*
factor ⟶ *number*
factor ⟶ - *factor*
factor ⟶ (*expr*)
add_op ⟶ +
add_op ⟶ -
mul_op ⟶ *
mul_op ⟶ /

†Actually, it doesn't have to be the *next* token. It could be the *next k* tokens. However, *k* has to be a constant, so ambiguous cases can always be constructed.

[*Eliminating Left Recursion*]

- Rule sets of the form
  - $X \longrightarrow X\alpha$ (where α does not start with X)
  - $X \longrightarrow \beta$

  generate strings of the form βα*.
- This can be replaced by
  - $X \longrightarrow \beta X'$
  - $X' \longrightarrow \alpha X'$
  - $X' \longrightarrow \varepsilon$

  moving the recursion from the left side to the right side.

This method isn't restricted to a single production. For example,

$$\begin{array}{l} X \longrightarrow X\alpha_1 \\ X \longrightarrow X\alpha_2 \\ X \longrightarrow X\alpha_3 \\ X \longrightarrow \beta_1 \\ X \longrightarrow \beta_2 \\ X \longrightarrow \beta_3 \end{array} \Rightarrow \begin{array}{l} X \longrightarrow \beta_1 X' \\ X \longrightarrow \beta_2 X' \\ X \longrightarrow \beta_3 X' \\ X' \longrightarrow \alpha_1 X' \\ X' \longrightarrow \alpha_2 X' \\ X' \longrightarrow \alpha_3 X' \\ X' \longrightarrow \varepsilon \end{array}$$

Predictive Parsing Example

- Eliminate left recursion by rewriting the *expr* and *term* production rules.
- We introduce non-terminals *expr1* and *term1* to handle the right recursion.

expr ⟶ *term expr1*
expr1 ⟶ *add_op term expr1*
expr1 ⟶ ε

term ⟶ *factor term1*
term1 ⟶ *mul_op factor term1*
term1 ⟶ ε

factor ⟶ *id*
factor ⟶ *number*
factor ⟶ - *factor*
factor ⟶ (*expr*)
add_op ⟶ +
add_op ⟶ -
mul_op ⟶ *
mul_op ⟶ /

Predictive Parsing Example

- Let's number the final set of production rules to make them easier to refer to as we continue the processing.

1. *expr* ⟶ *term expr1*
2. *expr1* ⟶ *add_op term expr1*
3. *expr1* ⟶ ε
4. *term* ⟶ *factor term1*
5. *term1* ⟶ *mul_op factor term1*
6. *term1* ⟶ ε
7. *factor* ⟶ *id*
8. *factor* ⟶ *number*
9. *factor* ⟶ - *factor*
10. *factor* ⟶ (*expr*)
11. *add_op* ⟶ +
12. *add_op* ⟶ -
13. *mul_op* ⟶ *
14. *mul_op* ⟶ /

Predictive Parsing Example

- To predict which production rule to use, we have to know three items for each non-terminal *X*.

- NULLABLE(*X*): Does *X* ever derive ε?
- FIRST(*X*): Which terminals can appear *first* in a string derived from *X*?
- FOLLOW(*X*): Which terminals can appear *immediately after X*?

1. *expr* ⟶ *term expr1*
2. *expr1* ⟶ *add_op term expr1*
3. *expr1* ⟶ ε
4. *term* ⟶ *factor term1*
5. *term1* ⟶ *mul_op factor term1*
6. *term1* ⟶ ε
7. *factor* ⟶ *id*
8. *factor* ⟶ *number*
9. *factor* ⟶ - *factor*
10. *factor* ⟶ (*expr*)
11. *add_op* ⟶ +
12. *add_op* ⟶ -
13. *mul_op* ⟶ *
14. *mul_op* ⟶ /

[*NULLABLE*]

- Computing NULLABLE(*X*) is a ~~tedious~~ straightforward though iterative process.
- For our grammar, the only NULLABLE items are *expr1* and *term1*.
  - They derive ε directly.
  - No other non-terminal derives either of these non-terminals without also including something else.

```
for each terminal and non-terminal X
  NULLABLE[X] ⟵ False

repeat
  for each rule X ⟶ Y1 Y2 … Yk
    if k = 0
      NULLABLE[X] ⟵ True

    else if Y1 Y2 … Yk are all NULLABLE
      NULLABLE[X] ⟵ True

until NULLABLE didn't change.
```

[*FIRST*]

- As with NULLABLE, computing FIRST(*X*) is an ~~incredibly tedious~~ straightforward though iterative process.
- After iterating, for a while, we find these FIRST sets.

FIRST(add_op) = { '+', '-' }
FIRST(mul_op) = { '*', '/' }
FIRST(factor) = { id, number, '-', '(' }
FIRST(term1) = { '*', '/' }
FIRST(term) = { id, number, '-', '(' }
FIRST(expr1) = { '+', '-' }
FIRST(expr) = { id, number, '-', '(' }

```
for each terminal X
  FIRST[X] ⟵ { X }

for each non-terminal X
  FIRST[X] ⟵ {}

repeat
  for each rule X ⟶ Y1 Y2 … Yk
    for i ⟵ 1 .. k
      if i = 1
        FIRST[X] ⟵ FIRST[X] ∪ FIRST[Yi]

      elif Y1 Y2 … Yi-1 are all NULLABLE
        FIRST[X] ⟵ FIRST[X] ∪ FIRST[Yi]

until FIRST didn't change.
```

[*FOLLOW*]

- Yes, computing FOLLOW(*X*) is a~~n incredibly, mind-bogglingly tedious~~ straightforward though iterative process.
- After iterating for a while, we find these FOLLOW sets.

FOLLOW(add_op) = { id, number, '-', '(' }
FOLLOW(mul_op) = { id, number, '-', '(' }
FOLLOW(factor) = { '*', '/', '+', '-', ')', EOF }
FOLLOW(term1) = { '+', '-', ')', EOF }
FOLLOW(term) = { '+', '-', ')', EOF }
FOLLOW(expr1) = { ')', EOF }
FOLLOW(expr) = { ')', EOF }

```
for each terminal and non-terminal X
  FOLLOW[X] ← {}

repeat
  for each rule X → Y1 Y2 … Yk
    for i ← 1 .. k
      if i = k
        FOLLOW[Yi] ← FOLLOW[Yi] ∪ FOLLOW[X]

      elif Yi+1 … Yk are all NULLABLE
        FOLLOW[Yi] ← FOLLOW[Yi] ∪ FOLLOW[X]

      for j ← i+1 .. k
        if i+1 = j
          FOLLOW[Yi] ← FOLLOW[Yi] ∪ FIRST[Yj]
        elif Yi+1 … Yj-1 are all NULLABLE
          FOLLOW[Yi] ← FOLLOW[Yi] ∪ FIRST[Yj]

until FOLLOW didn't change.
```

Predictive Parsing Example

- Given the NULLABLE and FIRST and FOLLOW sets, we can now construct a *parse table* for our CFG.
- This table tells us which rule to use when we are trying to parse a given *non-terminal* and are seeing a given *terminal*.
- We have one row for each *non-terminal* and one column for each *terminal*.

```
for each terminal X
  for each non-terminal Y
    TABLE[X,Y] ← {}

for each rule X → δ
  for each T ∈ FIRST[δ]
    TABLE[X,T] ← TABLE[X,T] ∪ {X→δ}

  if NULLABLE[δ]
    for each T ∈ FOLLOW[X]
      TABLE[X,T] ← TABLE[X,T] ∪ {X→δ}
```

Predictive Parsing Example

- For our CFG, we get this parse table.

	+	-	*	/	id	number	(	)	EOF
add_op	11	12							
mul_op			13	14					
factor		9			7	8	10		
term1	6	6	5	5				6	6
term		4			4	4	4		
expr1	2	2						3	3
expr		1			1	1	1		

Predictive Parsing Example

- For example, if we are trying to parse an *expr* and we are seeing the (token, we use production rule **1**.
- Any box that does not have a rule number indicates an error condition for that non-terminal / terminal combination.
  - E.g., trying to parse a *mul_op* and seeing the - token is an error.
- Since no box has more than one rule, we are confirmed in our thinking that the grammar is *unambiguous*.

Predictive Parsing Example

- Great, we have the parse table. How do we get a parser?
- It's not difficult. We just write a routine for each non-terminal.
- The parse table tells us what to do in each case.
  - At right are two of the seven required routines.

```
def EXPR() :
  token = peekToken()

  if token in FIRST_EXPR :
    value = TERM() + EXPR1()

  else :
    print( 'Error!  EXPR saw %s when expecting %s.'
      % ( token, FIRST_EXPR ) )
    raise ValueError

  return value

def EXPR1() :
  token = peekToken()

  if token in [ PLUS, MINUS ] :
    value = ADD_OP() + TERM() + EXPR1()

  elif token in [ RPAREN, EOF ] :
    value = 'ε'

  else :
    print( 'Error!  EXPR1 saw %s when expecting %s.'
      % ( token, FIRST_EXPR1 ) )
    raise ValueError

  return value
```

Predictive Parsing Example

- For each non-terminal, we *peek* at the next token and then take the action recorded in the parse table.
- E.g., if `EXPR()` sees `-`, `id`, `number`, or `(`, it knows it's supposed to use rule 1, *expr* ⟶ *term expr1*.
- It therefore calls `TERM()` and then `EXPR1()`.
- Anything else is an error.

```
def EXPR() :
  token = peekToken()

  if token in FIRST_EXPR :
    value = TERM() + EXPR1()

  else :
    print( 'Error!  EXPR saw %s when expecting %s.'
      % ( token, FIRST_EXPR ) )
    raise ValueError

  return value

def EXPR1() :
  token = peekToken()

  if token in [ PLUS, MINUS ] :
    value = ADD_OP() + TERM() + EXPR1()

  elif token in [ RPAREN, EOF ] :
    value = 'ε'

  else :
    print( 'Error!  EXPR1 saw %s when expecting %s.'
      % ( token, FIRST_EXPR1 ) )
    raise ValueError

  return value
```

Predictive Parsing Example

- Eventually we get to routines that have to *consume* tokens after peeking at them.
- `advanceToken()` moves to the next token.
- `eat()` ensures that the token matches its argument, then moves across it.

```
def FACTOR( indent ) :
  token = peekToken()

  if token in [ ID, NUMBER ] :
    advanceToken()
    value = token

  elif token == MINUS :
    advanceToken()
    value = MINUS + FACTOR()

  elif token == LPAREN :
    advanceToken()
    value = LPAREN + EXPR() + RPAREN
    eat( RPAREN )

  else :
    print( 'Error!  FACTOR saw %s when expecting %s.'
      % ( token, FIRST_FACTOR ) )
    raise ValueError

  return value
```

Predictive Parsing Example

- This is known as a *recursive descent* parser.
- Writing this kind of parser by hand is very common …
  - When the grammar is simple!
- … which is the main reason we spend the time exploring the method.

```
def FACTOR( indent ) :
  token = peekToken()

  if token in [ ID, NUMBER ] :
    advanceToken()
    value = token

  elif token == MINUS :
    advanceToken()
    value = MINUS + FACTOR()

  elif token == LPAREN :
    advanceToken()
    value = LPAREN + EXPR() + RPAREN
    eat( RPAREN )

  else :
    print( 'Error!  FACTOR saw %s when expecting %s.'
      % ( token, FIRST_FACTOR ) )
    raise ValueError

  return value
```

[*Table-Driven Predictive Parsing*]

- By the way, we do not have to hand-code the parser.
- A general method exists that parses *directly* from the table.
- Simpler than using hand-coded routines, but can be inefficient.
- (*If we're going to use a generator, why not a more powerful one?*)

```
push( startSymbol )
token ← readNextToken()

repeat
  X ← pop()

  if terminal( X ) or X = EOF
    if X = token
      token ← readNextToken()
    else
      // Needed a token and current one didn't match.
      error()

  elif TABLE[ X, token ] is empty
    // No rule for this non-terminal / token pair.
    error()

  else
    // Have a rule to use. Push its RHS onto
    // the stack in reverse order.
    for Y in reverse( RHS( TABLE[ X, token ] ) )
      push( Y )

until X = EOF
```

Predictive Parsing Example

- And, the parser *really*, *really* works!

```
#----------
Trying ['number', '-', 'number', 'EOF'] ...
FACTOR:     number
TERM1 :     ε
TERM  :   numberε
ADD_OP:     -
FACTOR:       number
TERM1 :       ε
TERM  :     numberε
EXPR1 :     ε
EXPR1 :   -numberεε
EXPR  : numberε-numberεε

Success!
  numberε-numberεε
```

```
#----------
Trying ['number', '+', 'number',
        '*', 'number', 'EOF'] ...
FACTOR:     number
TERM1 :     ε
TERM  :   numberε
ADD_OP:     +
FACTOR:       number
MUL_OP:         *
FACTOR:         number
TERM1 :         ε
TERM1 :       *numberε
TERM  :     number*numberε
EXPR1 :     ε
EXPR1 :   +number*numberεε
EXPR  : numberε+number*numberεε

Success!
  numberε+number*numberεε
```

```
#----------
Trying ['number', '/', 'number',
        '-', 'number', 'EOF'] ...
FACTOR:     number
MUL_OP:       /
FACTOR:       number
TERM1 :       ε
TERM1 :     /numberε
TERM  :   number/numberε
ADD_OP:     -
FACTOR:       number
TERM1 :       ε
TERM  :     numberε
EXPR1 :     ε
EXPR1 :   -numberεε
EXPR  : number/numberε-numberεε

Success!
  number/numberε-numberεε
```

Predictive Parsing Example

- Even with *errors* and *complex* cases!

```
#----------
Trying [')', 'EOF'] ...
Error!  EXPR saw ) when expecting ['-', 'id', 'number', '('].
Parse error!

#----------
Trying ['-', '*', 'EOF'] ...
Error!  FACTOR saw * when expecting ['-', 'id', 'number', '('].
Parse error!

#----------
Trying ['number', '-', '+', 'EOF'] ...
FACTOR:     number
TERM1 :     ε
TERM  :   numberε
ADD_OP:     -
Error!  TERM saw + when expecting ['-', 'id', 'number', '('].
Parse error!
```

```
#----------
Trying ['(', 'number', '+', 'number',
        ')', '*', 'number', 'EOF'] ...
FACTOR:           number
TERM1 :           ε
TERM  :         numberε
ADD_OP:           +
FACTOR:             number
TERM1 :             ε
TERM  :           numberε
EXPR1 :           ε
EXPR1 :         +numberεε
EXPR  :       numberε+numberεε
FACTOR:     (numberε+numberεε)
MUL_OP:       *
FACTOR:       number
TERM1 :       ε
TERM1 :     *numberε
TERM  :   (numberε+numberεε)*numberε
EXPR1 :   ε
EXPR  : (numberε+numberεε)*numberεε

Success!
  (numberε+numberεε)*numberεε
```

Predictive Parsing Example

- So what are those ε characters?
- That's where *expr1* or *term1* derived the empty string (ε).
  - Rules 3 and 6.
- Remember, we got rid of *left* recursion by converting it to *right* recursion.

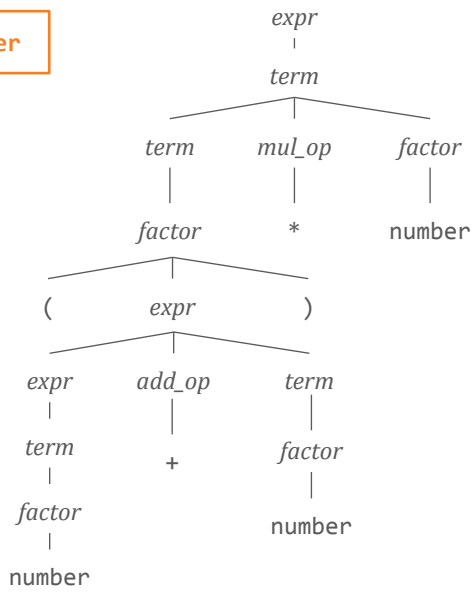
1. *expr* ⟶ *term expr1*
2. *expr1* ⟶ *add_op term expr1*
3. *expr1* ⟶ ε
4. *term* ⟶ *factor term1*
5. *term1* ⟶ *mul_op factor term1*
6. *term1* ⟶ ε
7. *factor* ⟶ *id*
8. *factor* ⟶ *number*
9. *factor* ⟶ - *factor*
10. *factor* ⟶ (*expr*)
11. *add_op* ⟶ +
12. *add_op* ⟶ -
13. *mul_op* ⟶ *
14. *mul_op* ⟶ /

Predictive Parsing Example

- Well, that was *painful*.
- Also, the eventual parsing we got from the `LL(1)` method is not exactly what we might have expected.
  - We had to change our grammar to eliminate left recursion.
- Did you notice the weird way the productions came out?

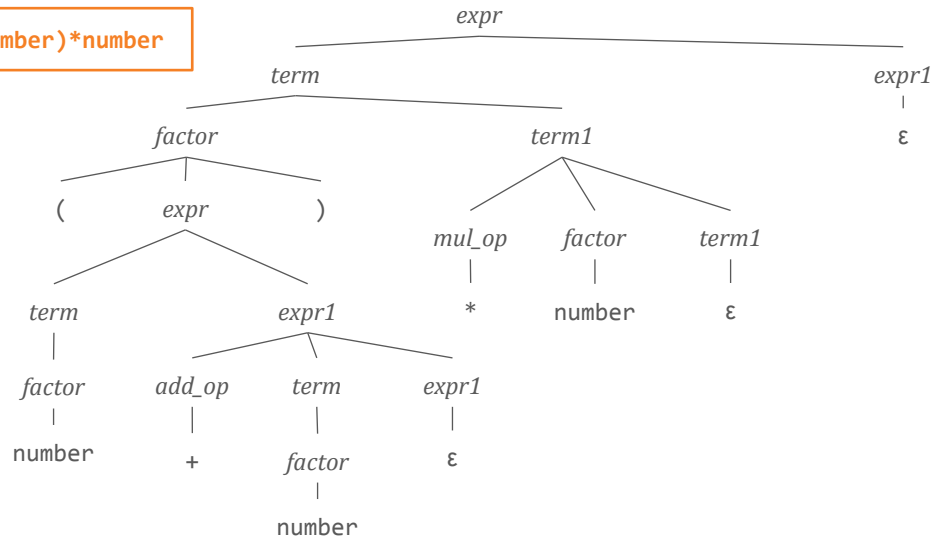
Expected Parse Tree for (number+number)*number

`(number+number)*number`



Actual Parse Tree

`(number+number)*number`



Predictive Parsing Example

- Pretty awful, huh?
- Well, no matter, we can always fix it up in the action routines.
- *But* … do we *want* to?
- ***Again***, there has *got* to be a better way!

LR(*k*) Parsing

- There is a better way.
- It's called **LR(*k*)** parsing.

- LR(*k*) ≡ ***L****eft-to-right* scan of the input, producing a ***R****ightmost* derivation, using ***k*** tokens of lookahead.

LL(1) vs LR(*k*) Parsing

- The LL(1) Predictive Parsing that we did was *top-down*.
  - We used the production rules from *left* to *right*.

- LR(*k*) parsing is *bottom-up*.
  - We use the production rules from *right* to *left*.

- As with the table-driven predictive parsing, we use a *stack* with LR(*k*) parsing.
  - But we use the stack in a more sophisticated way.

LR(*k*) Parsing

- We use a *stack* and the *input stream* to decide what to do.
- The first *k* tokens in the input stream form the *lookahead*.
- Depending on the stack and the lookahead, the parser takes one of two possible actions:
  - > *Shift* : Push the first input token onto the stack.
  - > *Reduce* : Select some grammar rule, e.g., $X \longrightarrow A\ B\ C$; pop *C*, *B*, *A* off the stack; and push *X* onto the stack.

LR(*k*) Parsing

- Initially the *stack* is empty and the *lookahead* is the first *k* tokens from the *input stream*.
- If the parser makes it to the point where it can *shift* the *EOF* marker onto the stack, it has *accepted* the input as valid.

LR(*k*) Parsing

- The decision to *reduce* is made based on the top few items on the stack, which need to match the RHS of a production rule. Three cases can occur.
- ① The top few items do not match the RHS of *any* rule of the grammar.
  - This is *easy* to resolve.
  - Just *shift* another token from the input stream onto the stack and look again.

LR(*k*) Parsing

- ② The top few items match the RHS of *one* rule of the grammar.
  - This is *easy* to resolve as well.
  - Just *reduce* by popping off the stack items corresponding to the RHS of the rule.
    - The popped items are known as a *handle*.
  - Then push the LHS of the rule (a non-terminal) onto the stack.

LR(*k*) Parsing

- ③ The top few items match the RHS of *multiple* rules of the grammar.
  - This is *not* so easy to resolve.
  - The grammar is *ambiguous* so we don't know which reduction to make.
  - This is known as a *reduce/reduce* ambiguity.

LR(*k*) Parsing

- So how do we find such issues in the grammar?
  - And are there other kinds of problems?

- We have to produce the ACTION and GOTO tables and see what problems show up along the way.
- Producing these tables is similar to what we did with *Predictive Parsing*.
- Aside from *reduce/reduce* conflicts, there are *shift/reduce* conflicts, which occur when it's not clear whether to shift a token onto the stack or to reduce what's already there.

LR(*k*) Parsing

- We analyze the grammar by building *item sets* based on the rules and how far we've gotten into a parse.
- For example, the *item*
  - `E -> E . + T`

  indicates we using the rule E -> E + T, we have seen an E, and that we are expecting a '+' token next.
- An *item set* is a collection of *items* that are all in the same state.

Analyzing an Example Grammar

- We'll start with a simplified version of the classic expression grammar.
  - Only one add op, +.
  - Only one mul op, *.
- It's unambiguous, so it shouldn't have any analysis problems.

```
S → E
E → E + T
E → T
T → T * F
T → F
F → id
F → ( E )
```

Analyzing an Example Grammar

```
Grammar

Rule 0     S' -> S
Rule 1     S -> E
Rule 2     E -> E + T
Rule 3     E -> T
Rule 4     T -> T * F
Rule 5     T -> F
Rule 6     F -> id
Rule 7     F -> ( E )
```

```
Terminals, with rules where they appear

(                    : 7
*                    : 4
+                    : 2
)                    : 7
error                :
id                   : 6

Nonterminals, with rules where they appear

E                    : 1 2 7
F                    : 4 5
S                    : 0
T                    : 2 3 4
```

Analyzing an Example Grammar

```
state 0

    (0) S' -> . S
    (1) S -> . E
    (2) E -> . E + T
    (3) E -> . T
    (4) T -> . T * F
    (5) T -> . F
    (6) F -> . id
    (7) F -> . ( E )

    id              shift and go to state 5
    (               shift and go to state 6

    S                           shift and go to state 1
    E                           shift and go to state 2
    T                           shift and go to state 3
    F                           shift and go to state 4
```

Analyzing an Example Grammar

```
state 1

    (0) S' -> S .


state 2

    (1) S -> E .
    (2) E -> E . + T

    $end            reduce using rule 1 (S -> E .)
    +               shift and go to state 7
```

Analyzing an Example Grammar

```
state 3

    (3) E -> T .
    (4) T -> T . * F

    +         reduce using rule 3 (E -> T .)
    $end      reduce using rule 3 (E -> T .)
    )         reduce using rule 3 (E -> T .)
    *         shift and go to state 8


state 4

    (5) T -> F .

    *         reduce using rule 5 (T -> F .)
    +         reduce using rule 5 (T -> F .)
    $end      reduce using rule 5 (T -> F .)
    )         reduce using rule 5 (T -> F .)
```

Analyzing an Example Grammar

```
state 5

    (6) F -> id .

    *        reduce using rule 6 (F -> id .)
    +        reduce using rule 6 (F -> id .)
    $end     reduce using rule 6 (F -> id .)
    )        reduce using rule 6 (F -> id .)
```

Analyzing an Example Grammar

```
state 6

    (7) F -> ( . E )
    (2) E -> . E + T
    (3) E -> . T
    (4) T -> . T * F
    (5) T -> . F
    (6) F -> . id
    (7) F -> . ( E )

    id          shift and go to state 5
    (           shift and go to state 6

    E                               shift and go to state 9
    T                               shift and go to state 3
    F                               shift and go to state 4
```

Analyzing an Example Grammar

```
state 7

    (2) E -> E + . T
    (4) T -> . T * F
    (5) T -> . F
    (6) F -> . id
    (7) F -> . ( E )

    id          shift and go to state 5
    (           shift and go to state 6

    T                               shift and go to state 10
    F                               shift and go to state 4
```

Analyzing an Example Grammar

```
state 8

    (4) T -> T * . F
    (6) F -> . id
    (7) F -> . ( E )

    id          shift and go to state 5
    (           shift and go to state 6

    F                               shift and go to state 11

state 9

    (7) F -> ( E . )
    (2) E -> E . + T

    )           shift and go to state 12
    +           shift and go to state 7
```

Analyzing an Example Grammar

```
state 10

    (2) E -> E + T .
    (4) T -> T . * F

    +               reduce using rule 2 (E -> E + T .)
    $end            reduce using rule 2 (E -> E + T .)
    )               reduce using rule 2 (E -> E + T .)
    *               shift and go to state 8


state 11

    (4) T -> T * F .

    *               reduce using rule 4 (T -> T * F .)
    +               reduce using rule 4 (T -> T * F .)
    $end            reduce using rule 4 (T -> T * F .)
    )               reduce using rule 4 (T -> T * F .)
```

Analyzing an Example Grammar

```
state 12

    (7) F -> ( E ) .

    *               reduce using rule 7 (F -> ( E ) .)
    +               reduce using rule 7 (F -> ( E ) .)
    $end            reduce using rule 7 (F -> ( E ) .)
    )               reduce using rule 7 (F -> ( E ) .)
```

Analyzing an Example Grammar

Action Table

```
 0: {'id': 5, '(': 6},
 1: {'$': 0},
 2: {'$': -1, '+': 7},
 3: {'$': -3, '+': -3, ')': -3, '*': 8},
 4: {'$': -5, '+': -5, ')': -5, '*': -5},
 5: {'$': -6, '+': -6, ')': -6, '*': -6},
 6: {'id': 5, '(': 6},
 7: {'id': 5, '(': 6},
 8: {'id': 5, '(': 6},
 9: {'+': 7, ')': 12},
10: {'$': -2, '+': -2, ')': -2, '*': 8},
11: {'$': -4, '+': -4, ')': -4, '*': -4},
12: {'$': -7, '+': -7, ')': -7, '*': -7},
```

Goto Table

```
0: {'S': 1, 'E': 2, 'T': 3, 'F': 4},
6: {'E': 9, 'T': 3, 'F': 4},
7: {'T': 10, 'F': 4},
8: {'F': 11}
```

Analyzing an Example Grammar

- This example grammar is straightforward.
  - There are no ambiguities.
  - No shift/reduce conflicts.
  - No reduce/reduce conflicts.
- How boring.
- Let's try another grammar, this time with a problem.

Grammar with a Shift/Reduce Conflict

- Our grammar is simple, merely demonstrating how one might represent an `if` statement.
- We have two kinds of `if` statement, one with an `else` clause and one without.
- The ambiguity is obvious. Let's see what the analysis reports.

```
S → if E then S else S
S → if E then S
S → E
E → ID
```

Grammar with a Shift/Reduce Conflict

```
Grammar

Rule 0     S' -> S
Rule 1     S -> IF E THEN S ELSE S
Rule 2     S -> IF E THEN S
Rule 3     S -> E
Rule 4     E -> ID
```

```
Terminals, with rules where they appear

ELSE                 : 1
ID                   : 4
IF                   : 1 2
THEN                 : 1 2
error                :

Nonterminals, with rules where they appear

E                    : 1 2 3
S                    : 1 1 2 0
```

Grammar with a Shift/Reduce Conflict

```
state 0

    (0) S' -> . S
    (1) S -> . IF E THEN S ELSE S
    (2) S -> . IF E THEN S
    (3) S -> . E
    (4) E -> . ID

    IF              shift and go to state 2
    ID              shift and go to state 4

    S                               shift and go to state 1
    E                               shift and go to state 3
```

Grammar with a Shift/Reduce Conflict

```
state 1

    (0) S' -> S .


state 2

    (1) S -> IF . E THEN S ELSE S
    (2) S -> IF . E THEN S
    (4) E -> . ID

    ID              shift and go to state 4

    E                               shift and go to state 5
```

Grammar with a Shift/Reduce Conflict

```
state 3

    (3) S -> E .

    $end            reduce using rule 3 (S -> E .)
    ELSE            reduce using rule 3 (S -> E .)


state 4

    (4) E -> ID .

    $end            reduce using rule 4 (E -> ID .)
    THEN            reduce using rule 4 (E -> ID .)
    ELSE            reduce using rule 4 (E -> ID .)
```

Grammar with a Shift/Reduce Conflict

```
state 5

    (1) S -> IF E . THEN S ELSE S
    (2) S -> IF E . THEN S

    THEN            shift and go to state 6

state 6

    (1) S -> IF E THEN . S ELSE S
    (2) S -> IF E THEN . S
    (1) S -> . IF E THEN S ELSE S
    (2) S -> . IF E THEN S
    (3) S -> . E
    (4) E -> . ID

    IF              shift and go to state 2
    ID              shift and go to state 4

    E                           shift and go to state 3
    S                           shift and go to state 7
```

Grammar with a Shift/Reduce Conflict

```
state 7

    (1) S -> IF E THEN S . ELSE S
    (2) S -> IF E THEN S .

  ! shift/reduce conflict for ELSE resolved as shift
    ELSE            shift and go to state 8
    $end            reduce using rule 2 (S -> IF E THEN S .)

  ! ELSE            [ reduce using rule 2 (S -> IF E THEN S .) ]
```

Grammar with a Shift/Reduce Conflict

```
state 8

    (1) S -> IF E THEN S ELSE . S
    (1) S -> . IF E THEN S ELSE S
    (2) S -> . IF E THEN S
    (3) S -> . E
    (4) E -> . ID

    IF              shift and go to state 2
    ID              shift and go to state 4

    E                               shift and go to state 3
    S                               shift and go to state 9
```

Grammar with a Shift/Reduce Conflict

```
state 9

    (1) S -> IF E THEN S ELSE S .

    $end            reduce using rule 1 (S -> IF E THEN S ELSE S .)
    ELSE            reduce using rule 1 (S -> IF E THEN S ELSE S .)

WARNING:
WARNING: Conflicts:
WARNING:
WARNING: shift/reduce conflict for ELSE in state 7 resolved as shift
```

Grammar with a Shift/Reduce Conflict

Action Table

```
0: { 'IF': 2, 'ID': 4 }
1: { '$end': 0 }
2: { 'ID': 4 }
3: { '$end': -3, 'ELSE': -3 }
4: { '$end': -4, 'ELSE': -4, 'THEN': -4 }
5: { 'THEN': 6 }
6: { 'IF': 2, 'ID': 4 }
7: { '$end': -2, 'ELSE': 8 }
8: { 'IF': 2, 'ID': 4 }
9: { '$end': -1, 'ELSE': -1 }
```

Goto Table

```
0: { 'S': 1, 'E': 3 }
2: { 'E':  5 }
6: { 'S': 7, 'E': 3 }
8: { 'S': 9, 'E': 3 }
```

Grammar with a Shift/Reduce Conflict

- OK, so there was a shift/reduce conflict.
- What do we do about?
- First, we have to decide what the correct, expected parse is supposed to be.
  - Normally, this would be to have the `else` clause bind with the closest available `if` that doesn't already have an `else` clause.
- Guess what? That's exactly what happens if we just resolve the shift/reduce conflict by *always* shifting!
  - That's the *default* behavior of the parser generator anyway.

Grammar with a Shift/Reduce Conflict

- So that wasn't very exciting after all.
- We could have rewritten the grammar so that this shift/reduce conflict doesn't occur, but why?
  - Or, we could have introduced a terminating marker for the `if` statement (as, e.g., Ada has, `end if`).
- OK, let's try another grammar, this one with *lots* of shift/reduce conflicts.

Another Grammar with Shift/Reduce Conflicts

- Another expression grammar, this time with multiple binary operators and no indication of precedence or associativity.
- The ambiguity is obvious. Let's see what the analysis reports.

```
E → INTEGER
E → E + E
E → E - E
E → E * E
E → E / E
E → ( E )
```

Another Grammar with Shift/Reduce Conflicts

```
Grammar

Rule 0     S' -> E
Rule 1     E -> INTEGER
Rule 2     E -> E PLUS E
Rule 3     E -> E MINUS E
Rule 4     E -> E MULTIPLY E
Rule 5     E -> E DIVIDE E
Rule 6     E -> LPAREN E RPAREN
```

```
Terminals, with rules where they appear

DIVIDE               : 5
INTEGER              : 1
LPAREN               : 6
MINUS                : 3
MULTIPLY             : 4
PLUS                 : 2
RPAREN               : 6
error                :

Nonterminals, with rules where they appear

E                    : 2 2 3 3 4 4 5 5 6 0
```

Another Grammar with Shift/Reduce Conflicts

```
state 0

    (0) S' -> . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                               shift and go to state 1
```

Another Grammar with Shift/Reduce Conflicts

```
state 1

    (0) S' -> E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7
```

Another Grammar with Shift/Reduce Conflicts

```
state 2

    (1) E -> INTEGER .

    PLUS            reduce using rule 1 (E -> INTEGER .)
    MINUS           reduce using rule 1 (E -> INTEGER .)
    MULTIPLY        reduce using rule 1 (E -> INTEGER .)
    DIVIDE          reduce using rule 1 (E -> INTEGER .)
    $end            reduce using rule 1 (E -> INTEGER .)
    RPAREN          reduce using rule 1 (E -> INTEGER .)
```

Another Grammar with Shift/Reduce Conflicts

```
state 3

    (6) E -> LPAREN . E RPAREN
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                           shift and go to state 8
```

Another Grammar with Shift/Reduce Conflicts

```
state 4

    (2) E -> E PLUS . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                           shift and go to state 9
```

Another Grammar with Shift/Reduce Conflicts

```
state 5

    (3) E -> E MINUS . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                           shift and go to state 10
```

Another Grammar with Shift/Reduce Conflicts

```
state 6

    (4) E -> E MULTIPLY . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                           shift and go to state 11
```

Another Grammar with Shift/Reduce Conflicts

```
state 7

    (5) E -> E DIVIDE . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                           shift and go to state 12
```

Another Grammar with Shift/Reduce Conflicts

```
state 8

    (6) E -> LPAREN E . RPAREN
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    RPAREN          shift and go to state 13
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7
```

Another Grammar with Shift/Reduce Conflicts

```
state 9

    (2) E -> E PLUS E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

  ! shift/reduce conflict for PLUS resolved as shift
  ! shift/reduce conflict for MINUS resolved as shift
  ! shift/reduce conflict for MULTIPLY resolved as shift
  ! shift/reduce conflict for DIVIDE resolved as shift
    $end            reduce using rule 2 (E -> E PLUS E .)
    RPAREN          reduce using rule 2 (E -> E PLUS E .)
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! PLUS            [ reduce using rule 2 (E -> E PLUS E .) ]
  ! MINUS           [ reduce using rule 2 (E -> E PLUS E .) ]
  ! MULTIPLY        [ reduce using rule 2 (E -> E PLUS E .) ]
  ! DIVIDE          [ reduce using rule 2 (E -> E PLUS E .) ]
```

Another Grammar with Shift/Reduce Conflicts

```
state 10

    (3) E -> E MINUS E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

  ! shift/reduce conflict for PLUS resolved as shift
  ! shift/reduce conflict for MINUS resolved as shift
  ! shift/reduce conflict for MULTIPLY resolved as shift
  ! shift/reduce conflict for DIVIDE resolved as shift
    $end            reduce using rule 3 (E -> E MINUS E .)
    RPAREN          reduce using rule 3 (E -> E MINUS E .)
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! PLUS            [ reduce using rule 3 (E -> E MINUS E .) ]
  ! MINUS           [ reduce using rule 3 (E -> E MINUS E .) ]
  ! MULTIPLY        [ reduce using rule 3 (E -> E MINUS E .) ]
  ! DIVIDE          [ reduce using rule 3 (E -> E MINUS E .) ]
```

Another Grammar with Shift/Reduce Conflicts

```
state 11

    (4) E -> E MULTIPLY E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

  ! shift/reduce conflict for PLUS resolved as shift
  ! shift/reduce conflict for MINUS resolved as shift
  ! shift/reduce conflict for MULTIPLY resolved as shift
  ! shift/reduce conflict for DIVIDE resolved as shift
    $end            reduce using rule 4 (E -> E MULTIPLY E .)
    RPAREN          reduce using rule 4 (E -> E MULTIPLY E .)
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! PLUS            [ reduce using rule 4 (E -> E MULTIPLY E .) ]
  ! MINUS           [ reduce using rule 4 (E -> E MULTIPLY E .) ]
  ! MULTIPLY        [ reduce using rule 4 (E -> E MULTIPLY E .) ]
  ! DIVIDE          [ reduce using rule 4 (E -> E MULTIPLY E .) ]
```

Another Grammar with Shift/Reduce Conflicts

```
state 12

    (5) E -> E DIVIDE E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

  ! shift/reduce conflict for PLUS resolved as shift
  ! shift/reduce conflict for MINUS resolved as shift
  ! shift/reduce conflict for MULTIPLY resolved as shift
  ! shift/reduce conflict for DIVIDE resolved as shift
    $end            reduce using rule 5 (E -> E DIVIDE E .)
    RPAREN          reduce using rule 5 (E -> E DIVIDE E .)
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! PLUS            [ reduce using rule 5 (E -> E DIVIDE E .) ]
  ! MINUS           [ reduce using rule 5 (E -> E DIVIDE E .) ]
  ! MULTIPLY        [ reduce using rule 5 (E -> E DIVIDE E .) ]
  ! DIVIDE          [ reduce using rule 5 (E -> E DIVIDE E .) ]
```

Another Grammar with Shift/Reduce Conflicts

```
state 13

    (6) E -> LPAREN E RPAREN .

    PLUS            reduce using rule 6 (E -> LPAREN E RPAREN .)
    MINUS           reduce using rule 6 (E -> LPAREN E RPAREN .)
    MULTIPLY        reduce using rule 6 (E -> LPAREN E RPAREN .)
    DIVIDE          reduce using rule 6 (E -> LPAREN E RPAREN .)
    $end            reduce using rule 6 (E -> LPAREN E RPAREN .)
    RPAREN          reduce using rule 6 (E -> LPAREN E RPAREN .)
```

Another Grammar with Shift/Reduce Conflicts

```
WARNING:
WARNING: Conflicts:
WARNING:
WARNING: shift/reduce conflict for PLUS in state 9 resolved as shift
WARNING: shift/reduce conflict for MINUS in state 9 resolved as shift
WARNING: shift/reduce conflict for MULTIPLY in state 9 resolved as shift
WARNING: shift/reduce conflict for DIVIDE in state 9 resolved as shift
WARNING: shift/reduce conflict for PLUS in state 10 resolved as shift
WARNING: shift/reduce conflict for MINUS in state 10 resolved as shift
WARNING: shift/reduce conflict for MULTIPLY in state 10 resolved as shift
WARNING: shift/reduce conflict for DIVIDE in state 10 resolved as shift
WARNING: shift/reduce conflict for PLUS in state 11 resolved as shift
WARNING: shift/reduce conflict for MINUS in state 11 resolved as shift
WARNING: shift/reduce conflict for MULTIPLY in state 11 resolved as shift
WARNING: shift/reduce conflict for DIVIDE in state 11 resolved as shift
WARNING: shift/reduce conflict for PLUS in state 12 resolved as shift
WARNING: shift/reduce conflict for MINUS in state 12 resolved as shift
WARNING: shift/reduce conflict for MULTIPLY in state 12 resolved as shift
WARNING: shift/reduce conflict for DIVIDE in state 12 resolved as shift
```

Another Grammar with Shift/Reduce Conflicts

- This grammar has ***16*** shift/reduce conflicts (4×4 because of the operators) and we ***can't*** just always shift.
  - This would cause precedence problems in some cases.
    - E.g., + might happen before *.
- The key word here is *precedence*. We could rewrite the grammar to separate precedence levels (as in that C excerpt), but why?
- Most compiler-compilers however allow the specification of operator precedence (and associativity).

Specifying Precedence and Associativity in `bison`

- Use the `%left` directive.
- Here we state that `TOKEN_PLUS` and `TOKEN_MINUS` are lower precedence than `TOKEN_SLAHS` and `TOKEN_STAR`.
  - They are all `left` associative.

```
%left TOKEN_MINUS TOKEN_PLUS
%left TOKEN_SLASH TOKEN_STAR
```

- (There're also `%right` and `%nonassoc` options for those kinds of operators.)

Another Grammar …

- Inserting the directives and rerunning `bison` results in *no* shift/reduce conflicts at all.
- We still have the same number / structure of states, but now we *know* when to shift and when to reduce so that precedence and associativity are honored.

Another Grammar …

```
Grammar

Rule 0     S' -> E
Rule 1     E -> INTEGER
Rule 2     E -> E PLUS E
Rule 3     E -> E MINUS E
Rule 4     E -> E MULTIPLY E
Rule 5     E -> E DIVIDE E
Rule 6     E -> LPAREN E RPAREN
```

```
Terminals, with rules where they appear

DIVIDE               : 5
INTEGER              : 1
LPAREN               : 6
MINUS                : 3
MULTIPLY             : 4
PLUS                 : 2
RPAREN               : 6
error                :

Nonterminals, with rules where they appear

E                    : 2 2 3 3 4 4 5 5 6 0
```

Another Grammar …

```
state 0

    (0) S' -> . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                               shift and go to state 1
```

Another Grammar …

```
state 1

    (0) S' -> E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7
```

Another Grammar …

```
state 2

    (1) E -> INTEGER .

    PLUS            reduce using rule 1 (E -> INTEGER .)
    MINUS           reduce using rule 1 (E -> INTEGER .)
    MULTIPLY        reduce using rule 1 (E -> INTEGER .)
    DIVIDE          reduce using rule 1 (E -> INTEGER .)
    $end            reduce using rule 1 (E -> INTEGER .)
    RPAREN          reduce using rule 1 (E -> INTEGER .)
```

Another Grammar …

```
state 3

    (6) E -> LPAREN . E RPAREN
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                           shift and go to state 8
```

Another Grammar …

```
state 4

    (2) E -> E PLUS . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                           shift and go to state 9
```

Another Grammar …

```
state 5

    (3) E -> E MINUS . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                           shift and go to state 10
```

Another Grammar …

```
state 6

    (4) E -> E MULTIPLY . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                           shift and go to state 11
```

Another Grammar …

```
state 7

    (5) E -> E DIVIDE . E
    (1) E -> . INTEGER
    (2) E -> . E PLUS E
    (3) E -> . E MINUS E
    (4) E -> . E MULTIPLY E
    (5) E -> . E DIVIDE E
    (6) E -> . LPAREN E RPAREN

    INTEGER         shift and go to state 2
    LPAREN          shift and go to state 3

    E                           shift and go to state 12
```

Another Grammar …

```
state 8

    (6) E -> LPAREN E . RPAREN
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    RPAREN          shift and go to state 13
    PLUS            shift and go to state 4
    MINUS           shift and go to state 5
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7
```

Another Grammar …

```
state 9

    (2) E -> E PLUS E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            reduce using rule 2 (E -> E PLUS E .)
    MINUS           reduce using rule 2 (E -> E PLUS E .)
    $end            reduce using rule 2 (E -> E PLUS E .)
    RPAREN          reduce using rule 2 (E -> E PLUS E .)
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! MULTIPLY        [ reduce using rule 2 (E -> E PLUS E .) ]
  ! DIVIDE          [ reduce using rule 2 (E -> E PLUS E .) ]
  ! PLUS            [ shift and go to state 4 ]
  ! MINUS           [ shift and go to state 5 ]
```

Another Grammar …

```
state 10

    (3) E -> E MINUS E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            reduce using rule 3 (E -> E MINUS E .)
    MINUS           reduce using rule 3 (E -> E MINUS E .)
    $end            reduce using rule 3 (E -> E MINUS E .)
    RPAREN          reduce using rule 3 (E -> E MINUS E .)
    MULTIPLY        shift and go to state 6
    DIVIDE          shift and go to state 7

  ! MULTIPLY        [ reduce using rule 3 (E -> E MINUS E .) ]
  ! DIVIDE          [ reduce using rule 3 (E -> E MINUS E .) ]
  ! PLUS            [ shift and go to state 4 ]
  ! MINUS           [ shift and go to state 5 ]
```

Another Grammar …

```
state 11

    (4) E -> E MULTIPLY E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            reduce using rule 4 (E -> E MULTIPLY E .)
    MINUS           reduce using rule 4 (E -> E MULTIPLY E .)
    MULTIPLY        reduce using rule 4 (E -> E MULTIPLY E .)
    DIVIDE          reduce using rule 4 (E -> E MULTIPLY E .)
    $end            reduce using rule 4 (E -> E MULTIPLY E .)
    RPAREN          reduce using rule 4 (E -> E MULTIPLY E .)

  ! PLUS            [ shift and go to state 4 ]
  ! MINUS           [ shift and go to state 5 ]
  ! MULTIPLY        [ shift and go to state 6 ]
  ! DIVIDE          [ shift and go to state 7 ]
```

Another Grammar …

```
state 12

    (5) E -> E DIVIDE E .
    (2) E -> E . PLUS E
    (3) E -> E . MINUS E
    (4) E -> E . MULTIPLY E
    (5) E -> E . DIVIDE E

    PLUS            reduce using rule 5 (E -> E DIVIDE E .)
    MINUS           reduce using rule 5 (E -> E DIVIDE E .)
    MULTIPLY        reduce using rule 5 (E -> E DIVIDE E .)
    DIVIDE          reduce using rule 5 (E -> E DIVIDE E .)
    $end            reduce using rule 5 (E -> E DIVIDE E .)
    RPAREN          reduce using rule 5 (E -> E DIVIDE E .)

  ! PLUS            [ shift and go to state 4 ]
  ! MINUS           [ shift and go to state 5 ]
  ! MULTIPLY        [ shift and go to state 6 ]
  ! DIVIDE          [ shift and go to state 7 ]
```

Another Grammar …

```
state 13

    (6) E -> LPAREN E RPAREN .

    PLUS            reduce using rule 6 (E -> LPAREN E RPAREN .)
    MINUS           reduce using rule 6 (E -> LPAREN E RPAREN .)
    MULTIPLY        reduce using rule 6 (E -> LPAREN E RPAREN .)
    DIVIDE          reduce using rule 6 (E -> LPAREN E RPAREN .)
    $end            reduce using rule 6 (E -> LPAREN E RPAREN .)
    RPAREN          reduce using rule 6 (E -> LPAREN E RPAREN .)
```

Another Grammar …

Action Table

```
0: {'INTEGER': 2, 'LPAREN': 3}
1: {'$end': 0, 'PLUS': 4, 'MINUS': 5,
    'MULTIPLY': 6, 'DIVIDE': 7}
2: {'$end': -1, 'PLUS': -1, 'MINUS': -1,
    'MULTIPLY': -1, 'DIVIDE': -1, 'RPAREN': -1}
3: {'INTEGER': 2, 'LPAREN': 3}
4: {'INTEGER': 2, 'LPAREN': 3}
5: {'INTEGER': 2, 'LPAREN': 3}
6: {'INTEGER': 2, 'LPAREN': 3}
7: {'INTEGER': 2, 'LPAREN': 3}
8: {'PLUS': 4, 'MINUS': 5, 'MULTIPLY': 6,
    'DIVIDE': 7, 'RPAREN': 13}
9: {'$end': -2, 'PLUS': -2, 'MINUS': -2,
    'MULTIPLY': 6, 'DIVIDE': 7, 'RPAREN': -2}
10: {'$end': -3, 'PLUS': -3, 'MINUS': -3,
     'MULTIPLY': 6, 'DIVIDE': 7, 'RPAREN': -3}
11: {'$end': -4, 'PLUS': -4, 'MINUS': -4,
     'MULTIPLY': -4, 'DIVIDE': -4, 'RPAREN': -4}
12: {'$end': -5, 'PLUS': -5, 'MINUS': -5,
     'MULTIPLY': -5, 'DIVIDE': -5, 'RPAREN': -5}
13: {'$end': -6, 'PLUS': -6, 'MINUS': -6,
     'MULTIPLY': -6, 'DIVIDE': -6, 'RPAREN': -6}
```

Goto Table

```
0: {'E': 1}
3: {'E': 8}
4: {'E': 9}
5: {'E': 10}
6: {'E': 11}
7: {'E': 12}
```

Another Grammar …

- Now there are no conflicts, precedence and associativity are properly honored, and the parse tree will be as expected.
- However, *shift/reduce* is not the only kind of conflict that can occur.
- There's also *reduce/reduce*.

Grammar with a Reduce/Reduce Conflict

- A contrived example grammar, but this is the kind of rule structure that leads to reduce/reduce conflicts.
- The ambiguity is obvious. Let's see what the analysis reports.

```
A → B c d
A → E c f
B → x y
E → x y
```

Grammar with a Reduce/Reduce Conflict

```
Grammar

Rule 0     S' -> A
Rule 1     A -> B c d
Rule 2     A -> E c f
Rule 3     B -> x y
Rule 4     E -> x y
```

```
Terminals, with rules where they appear

c                    : 1 2
d                    : 1
error                :
f                    : 2
x                    : 3 4
y                    : 3 4

Nonterminals, with rules where they appear

A                    : 0
B                    : 1
E                    : 2
```

Grammar with a Reduce/Reduce Conflict

```
state 0

    (0) S' -> . A
    (1) A -> . B c d
    (2) A -> . E c f
    (3) B -> . x y
    (4) E -> . x y

    x               shift and go to state 4

    A                               shift and go to state 1
    B                               shift and go to state 2
    E                               shift and go to state 3
```

Grammar with a Reduce/Reduce Conflict

```
state 1

    (0) S' -> A .


state 2

    (1) A -> B . c d

    c               shift and go to state 5


state 3

    (2) A -> E . c f

    c               shift and go to state 6
```

Grammar with a Reduce/Reduce Conflict

```
state 4

    (3) B -> x . y
    (4) E -> x . y

    y               shift and go to state 7


state 5

    (1) A -> B c . d

    d               shift and go to state 8
```

Grammar with a Reduce/Reduce Conflict

```
state 6

    (2) A -> E c . f

    f               shift and go to state 9


state 7

    (3) B -> x y .
    (4) E -> x y .

  ! reduce/reduce conflict for c resolved using rule 3 (B -> x y .)
    c               reduce using rule 3 (B -> x y .)

  ! c               [ reduce using rule 4 (E -> x y .) ]
```

Grammar with a Reduce/Reduce Conflict

```
state 8

    (1) A -> B c d .

    $end            reduce using rule 1 (A -> B c d .)


state 9

    (2) A -> E c f .

    $end            reduce using rule 2 (A -> E c f .)
```

Grammar with a Reduce/Reduce Conflict

```
WARNING:
WARNING: Conflicts:
WARNING:
WARNING: reduce/reduce conflict in state 7 resolved using rule (B -> x y)
WARNING: rejected rule (E -> x y) in state 7
WARNING: Rule (E -> x y) is never reduced
```

Grammar with a Reduce/Reduce Conflict

- To resolve the reduce/reduce conflict, the parser generator *arbitrarily* picked one of the rules and reduced by it.
- While letting a shift/reduce conflict be resolved by picking shift over reduce *might* be the proper resolution, letting the parser generator *arbitrarily* pick a rule to reduce by is ***not*** the proper way to resolve this sort of conflict.
- Reduce/reduce conflicts ***must always*** be investigated and ***resolved*** by refactoring / restating that part of the grammar.

Grammar with a Reduce/Reduce Conflict

- Fixing the contrived example is kind of pointless.
  - It's *contrived*.
- We'll look at reduce/reduce conflicts some more as we develop the grammar for our language.

A → B c d

A → E c f

B → x y

E → x y

Table-Driven Shift-Reduce Parsing

- The generalized *Shift-Reduce* parser is similar to the one for *Predictive Parsing*.
- We again stay in a loop looking at tokens and taking actions until we succeed or fail.
- Here, there are *three* kinds of entries in the `ACTION` table: *shift*, *reduce*, and *accept*.
- There's also a `GOTO` table for resetting the state after a reduce.

```
push(0)
token ← readNextToken()

loop
  s ← top()

  if ACTION[ s, token ] = 'si'
    push( i )     // Shift onto stack
    token ← readNextToken()

  elif ACTION[ s, token ] = 'ri'
    // Reduce by rule i: X → A1 A2 … An
    pop() n times
    s ← top()
    push( GOTO[ s, X ] )

  elif ACTION[ s, token ] = 'a'
    break     // Successful parse!

  else
    // Unexpected state / token pair.
    error()
end
```

LR(k) Parsing

- The parser decides between *shifting* and *reducing* via a DFA.
  - This DFA is not *parsing* the input as a DFA is too weak to parse a CFG.
  - The DFA is applied to the contents of the *stack*.

- The possible DFA actions are:
  - *Shift*(n) : Advance one token; push n onto the stack.
  - *Reduce*(m) : Pop as many items as the number of symbols on the RHS of CFG production rule m. For the state now on the top of the stack, look up the LHS of rule m to get the next state n. Push n onto the stack.
  - *Accept* : Stop parsing and report success.
  - *Error* : Stop parsing and report failure.

A Simple CFG …

1 $S \rightarrow S\ ;\ S$
2 $S \rightarrow \text{id}\ :=\ E$
3 $S \rightarrow \text{print}\ (\ L\ )$
4 $E \rightarrow \text{id}$
5 $E \rightarrow \text{num}$
6 $E \rightarrow E\ +\ E$
7 $E \rightarrow (S\ ,\ E\ )$
8 $L \rightarrow E$
9 $L \rightarrow L\ ,\ E$

A Simple CFG's LR Parsing Table

	id	num	print	;	,	+	:=	(	)	$	*S*	*E*	*L*
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9	s20	s10						s8				g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

s*n* Shift into state *n*.
g*n* Go to state *n*.
r*m* Reduce by rule *m*.
a Accept

Blanks indicate *error*.

LR Parsing Example

```
a := 7;
b := c + (d := 5 + 6, d)
```

Stack	*Input*	*Action*
1	a := 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id$_4$	:= 7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id$_4$:=$_6$	7 ; b := c + (d := 5 + 6 , d) $	*shift*
1 id$_4$:=$_6$ num$_{10}$	; b := c + (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ num
1 id$_4$:=$_6$ E_{11}	; b := c + (d := 5 + 6 , d) $	*reduce* $S \rightarrow$ id:=E
1 S_2	; b := c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$	b := c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$	:= c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$	c + (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ id$_{20}$	+ (d := 5 + 6 , d) $	*reduce* $E \rightarrow$ id
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11}	+ (d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$	(d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8	d := 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$	:= 5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$	5 + 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ num$_{10}$	+ 6 , d) $	*reduce* $E \rightarrow$ num
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11}	+ 6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11} +$_{16}$	6 , d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11} +$_{16}$ num$_{10}$	, d) $	*reduce* $E \rightarrow$ num
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11} +$_{16}$ E_{17}	, d) $	*reduce* $E \rightarrow E + E$
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 id$_4$:=$_6$ E_{11}	, d) $	*reduce* $S \rightarrow$ id:=E
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12}	, d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$	d) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$ id$_{20}$	) $	*reduce* $E \rightarrow$ id
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$ E_{21}	) $	*shift*
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ (8 S_{12} ,$_{18}$ E_{21})$_{22}$	$	*reduce* $E \rightarrow (S, E)$
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11} +$_{16}$ E_{17}	$	*reduce* $E \rightarrow E + E$
1 S_2 ;$_3$ id$_4$:=$_6$ E_{11}	$	*reduce* $S \rightarrow$ id:=E
1 S_2 ;$_3$ S_5	$	*reduce* $S \rightarrow S; S$
1 S_2	$	*accept*

LR(*k*) Parsing

- OK, enough theory.
- No, I am not going to drag you through how to process the CFG to get the DFA for parsing.
  - That's the job of a *compiler-compiler* tool such as `ply`, `yacc`, `bison`, etc.
  - FYI, at last check, Wikipedia's list of *Deterministic Context-Free Language Parser Generators* list had ***99*** entries.

2020 Feb 11, https://en.wikipedia.org/wiki/Comparison_of_parser_generators

The `bison` Parser Generator

- The point is that someone else has already done all of the work for you.
- Just use the `bison` parser generator so we can concentrate on the *language* instead of the tool that got us there.

`bison calc` Example

- A barebones “desk calculator”.
- Add +, Subtract -, Multiply *, and Divide / operators.
- Includes named variables to hold results.

```
calc > 1+1
2
calc > a = 1+2
calc > a
3
calc > a+3
6
calc > b = a*4
calc > b
12
calc > a+b
15
calc > c = a+b
calc > c
15
calc > d = e = f = 1
Syntax error at '='
```

https://github.com/dabeaz/ply

Parse Tree

- The output of the *Syntactic Analyzer* phase is a *Parse Tree*.
  - This tree represents the *structure* of the particular input token stream as determined by the language’s grammar rules.
  - *Unique* for a given stream of tokens.
    - If not, the grammar is ambiguous and needs to be fixed.
- It’s normally called a *Raw* (or *Concrete*) tree in that its leaves are the tokens and its internal nodes correspond to the applied production rules.

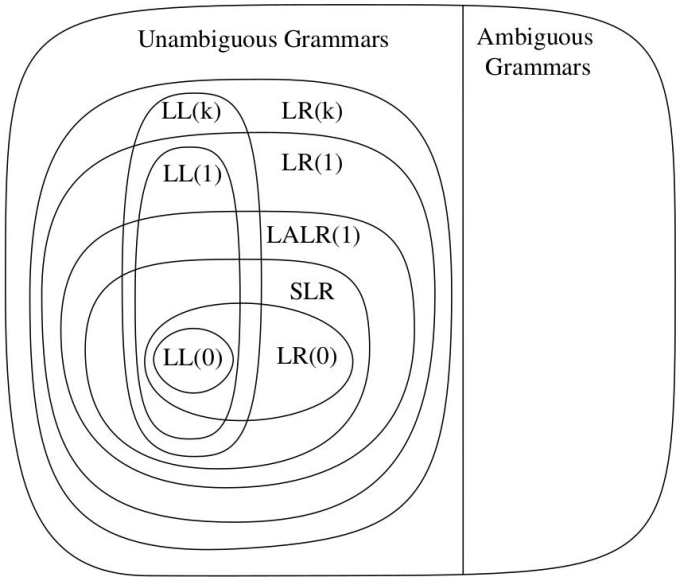
Parse Tree

- Correspondence is not always *exact*.
  - Not all tokens may be represented.
  - Alterations / Simplifications may have been made.
- There can be a *blurring* of the line between the *Parse Tree* and the *Abstract Syntax Tree* (*AST*).
  - ASTs will be considered in *Semantic Analysis*.

Syntactic Error Recovery

- What to do when a syntax error is detected?
  - Complain and die? — Most obvious answer but very unfriendly. Probably would like to see *other* errors.
  - However, have to avoid an avalanche of cascading errors.
- Exactly what's possible depends on how the parser was generated.
  - Different compiler compilers have (very) different error detection, control, and recovery mechanisms.
  - Hand-written parsers have the best flexibility!

Another Hierarchy of Grammar Classes

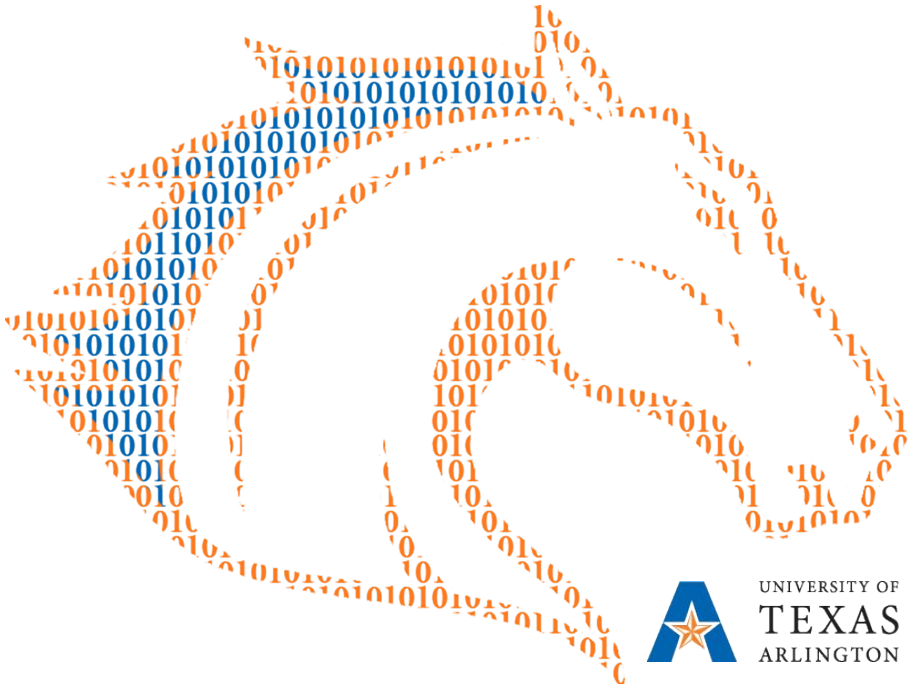


`LR` — (`ply`)

`LALR` — "LookAhead LR" merges some rules to get better memory usage. (`yacc` / Bison)

`SLR` — "Simple LR" merges still more rules.

Merging rules can cause Reduce / Reduce conflicts in some grammars.



UNIVERSITY OF TEXAS ARLINGTON
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Out[1]: (Show / Hide code)

`flex` Notes

CSE 4305 / CSE 5317 Compilers

2020 Fall Semester, Version 1.2, 2020 October 02

`flex` generates a C (C++ compatible) lexical analyzer based on a description of the token patterns to match and the code to execute when matches happen. It's a FOSS version of `lex`, developed separately from `lex` starting in about 1987 to get around the proprietary nature of `lex` and also to offer better performance.

An analyzer generated by `flex` scans the incoming characters using a Deterministic Finite Automaton created from all of the Regular Expression patterns given in the original description. Thus, in general a `flex` generated lexical analyzer achieves $\mathcal{O}(n)$ time complexity on matching. However, if one uses the `REJECT` capability of `flex` (a way to force the DFA to "try again" on a match that has already been made), greater than linear time complexity can result.

A Note on Versions

`flex` is remarkably stable — unlike a lot of crappy code that gets released nowadays.

Consequently, `flex` doesn't *need* to be released all that often as it does what it does quite well. Having said that, though, you should be using the most recent version `2.6.4`, released `2017-May-06`. That's over three years ago so there's no excuse not to have it.

I have received some questions from students re: warnings / errors from `flex` and in each case it's been because an older version of `flex` was being used. In one of the cases, a version from `2016-Mar-01` was being used, in another a version from *before* `2001-May-01` (?!?).

Everything in these *Notes* has been tested using `flex 2.6.4` in two development environments, with different processor architectures and kernel versions,

```
$ cat /etc/os-release
VERSION="18.04.5 LTS (Bionic Beaver)"
$ uname -a
Linux svr-test-01 4.15.0-118-generic #119-Ubuntu SMP Tue Sep 8 12:30:0
1 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
$ flex --version
flex 2.6.4
$
```

and

```
$ cat /etc/os-release
PRETTY_NAME="Raspbian GNU/Linux 10 (buster)"
$ uname -a
Linux icywits-01 5.4.51-v7+ #1333 SMP Mon Aug 10 16:45:19 BST 2020 arm
v7l GNU/Linux
$ flex --version
flex 2.6.4
$
```

That's no guarantee that an error hasn't crept in, though, so report any difficulties you might encounter, but please, please, please try it with `flex 2.6.4` before reporting.

Process

Generally,

- Put all of your `flex` input in a file named `<whatever>.l` , as described below.
- Generate the lexical analyzer by executing `flex <whatever>.l` .
- Include the resulting `lex.yy.c` file (the generated lexical analyzer) in the build of your entire program.

Simple, eh?

Format

The input file for `flex` should have a `.l` suffix. `flex` processes it line-by-line, collects the information it needs and generates the `lex.yy.c` lexical analyzer.

`flex` 's parsing of its input file is quite primitive so it's easy to make a typographical error that destroys `flex` 's understanding of what you're trying to do.

Be careful.

Some items have to be on lines by themselves, some items have to start at the left margin (no whitespace allowed in front), some items *must* be indented, etc., etc. We try to highlight these requirements in the following explanations.

One significant issue is `flex` 's processing of comments. It's safest to stick with C's `/* ... */` style of comments. You might get away with `//` style comments in various places, but it could take fiddling to figure out how to get `flex` to accept it. From the `flex` documentation,

> If you want to follow a simple rule, then always begin a comment on a new line, with one or more whitespace characters before the initial `/*` . This rule will work anywhere in the input file.

The `minimal` examples all work properly, so you can check their structure, formatting, etc. to see what's allowed by `flex` .

Input File Structure

A `.l` file is split into three sections and has this general format,

```
/* Definitions */

%%

/* Rules */

%%

/* User Code */
```

The `%%` marks have to be the only characters on their lines and at the left margin (that is, have no whitespace in front of them).

Definitions Section

The *Definitions Section* includes the specification of “definitions” (names for reusable units of regular expressions) and `%option` lines (options that control the functioning of `flex`). Also included is any C code the user wants included at the front of the generated lexical analyzer file. This C code is normally items that will be used by the action routines associated with the patterns given in the *Rules Section*.

Definitions

`flex`'s regular expression (RE) syntax (see the *Appendix: `flex` Regular Expressions* for more details on this syntax) does not support constructs such as `\d` (decimal digit ≡ `[0-9]`), `\s` (whitespace character ≡ `[ \f\n\r\t\v]`), `\w` (“word” character ≡ `[_a-zA-Z0-9]`), etc.

What `flex` *does* support is the creation of *Defined Names*. The definitions are put in the *Definitions Section* of the `.l` file. Here are some example definitions,

```
DIGIT          [0-9]
WHITE_SPACE    [ \f\n\r\t\v]
WORD_CHAR      [_a-zA-Z0-9]
HEX_DIGIT      [a-fA-F0-9]
```

The name part of the definition can have letters, digits, hyphens (`-`), and underscores (`_`), but it cannot start with a digit. It should be at the left margin (that is, no whitespace in front of the name).

In the *Rules Section* a defined name can be used in a pattern's RE by enclosing the name in braces `{...}`. So, for example, we could write the following,

```
/* Definitions Section */

DIGIT      [0-9]

%%

/ * Rules Section */

{DIGIT}+   {
    printf( "Saw integer %d.\n", atoi( yytext ) );
}
```

The `{ ... }` substitution can be used anywhere an RE can be used. The corresponding definition of the name (enclosed in parentheses `( ... )`) is substituted. The above example is equivalent to,

```
([0-9])+   {
    printf( "Saw integer %d.\n", atoi( yytext ) );
}
```

Note how the `{DIGIT}` name was replaced with `([0-9])`. The parentheses are automatically wrapped around the definition before doing the substitution to preserve the meaning of the definition no matter where it gets inserted.

Options

`%option` lines are used to inform `flex` as to how this `.l` file should be processed. Usually these options could have been specified on the `flex` command line, but putting them in the `.l` file is tidier.

There are zillions of `flex` options that you can read about in the `flex` documentation. Many are obscure and are relevant only in very special circumstances. We use a few in our `minimal` examples, so we'll explain them here.

- Header file

  ```
  %option header-file="lex.yy.h"
  ```

`flex` can generate a header file that exposes the external interface through which one can interact with the generated lexical analyzer. This is not important in the `minimal` examples as we put everything in the `.l` file but it will be important later when we generate `bison` parsers. This option says to generate that `.h` file with the given name.

> You might wonder why `flex` 's generated lexical analyzer is named `lex.yy.c` by default (and why the header file is named `lex.yy.h`). The best answer is “for historical reasons”. Ha!

- Not an interactive lexer

  ```
  %option never-interactive
  ```

Our `minimal` examples are not interactive—that is, input is not coming from an interactive session with a user. By setting this option, we let `flex` know this and `flex` can therefore generate a somewhat faster analyzer. (It has to do with how `flex` deals with lookahead.)

- A bunch of stuff we *don't* want

```
%option nodefault
%option noinput
%option nounistd
%option nounput
%option noyywrap
```

By default, `flex` includes a bunch of stuff in the generated lexical analyzer. If we're not using it, we don't want it. Therefore we turn *off* a bunch of items.

`nodefault` means `flex` should not generate the *default* rule, which is a rule that would match any character that isn't otherwise consumed by the given patterns. We don't need this default rule in the `minimal` examples because we have written an explicit mechanism to catch *illegal characters*. Even going beyond simple examples, you shouldn't let `flex` catch unexpected characters for you; your rules should match *anything* that can occur in the character stream, even if it's illegal. That help in generating meaningful error messages.

`noinput` and `nounput` mean `flex` should not generate the `input()` and `yyunput()` functions. We don't use them and if they are generated we're going to get `defined but not used` messages from `gcc` . (There are obscure cases where these two functions are useful, but we're not going to encounter them in an introductory example.)

`nounistd` means don't try to include the `<unistd.h>` header file. We don't need it and don't want it.

`noyywrap` means when we come to the end of the given input, we stop. There's no more input to process. (Complex scanning cases might involve multiple input files, but that's not going to come up in an introductory example.)

- Reports and warnings

```
%option perf-report perf-report
%option verbose verbose
%option warn
```

`flex` does a bunch of processing on your input. There are a number of statistics that can be displayed. It's useful to see them, especially the comments on anything that might be affecting the performance of generated lexical analyzer. The `perf-report` and `verbose` options are doubled because that causes `flex` to give more detailed reports.

`flex` can also detect certain issues with your input that are not really *errors* per se but would probably lead to unintended consequences. Specifying the `warn` option tells `flex` to warn you if these issues arise. (Why this option isn't *on* by default is beyond my comprehension.)

- Line numbering

```
%option yylineno
```

`flex` will count line numbers for you, but the capability is not turned on by default. This option enables line counting and makes the current line number available through the global variable `yylineno`.

> While `flex` will *count* lines when this option is enabled, it doesn't *initialize* `yylineno`, so don't forget to set `yylineno` to `1` in your `main` routine.

Included C Code

Sometimes you want to access certain C items inside the action routines associated with the patterns in the *Rules Section*. `flex` defines a bunch of useful items for you automatically. Others you will have to define yourself. You can put your C definitions in the *Definitions Section* inside `%{ ... %}` markers.

The format is,

```
%{
    /* Your C items */
%}
```

The `%{` and `%}` have to each be on their own lines and must be at the left margin (that is, no whitespace before them).

In the `minimal1` example, we include `<stdio.h>` (so we can do an `fprintf` in an action routine), define a token ID enumeration, define a spot to keep token values (`yylval`), and define a spot to keep the current column number (`yycolno`). Our included C code therefore looks like this,

```
%{
#include <stdio.h>

enum {
  tok_ID = 256,
  tok_INT_LIT,
};

union {
  int intval;
} yylval;

int yycolno;
%}
```

All of these items are threafter accessible in the *Rules Section* action routines.

Rules Section

The *Rules Section* includes the Regular Expressions (REs) that define the patterns of the token categories and the action routines that are associated with those patterns.

Every rule must be of the form,

```
regular-expression {
    action-routine
}
```

The *regular expression* must start at the left margin. No leading whitespace is allowed. The `{` starting the *action routine* must be on the same line as the regular expression.

For example, the following entry,

```
[0-9]+   {
    printf( "Saw digit string \"%s\", which has %d digit%s.",
      yytext, yyleng, yyleng == 1 ? "" : "s" );
}
```

has `[0-9]+` as its pattern. This regular expression matchs one-or-more (`+`) decimal digits (`[0-9]`). The action routine begins with the left brace `{` , *which must be on the same line as the regular expression*. This action routine merely prints a message showing the digit sequence that was matched as well as how many characters long it is.

`yytext` is a `char *` that points to a `NUL` -terminated string of the characters that were matched by the regular expression. `yyleng` is an `int` giving the length of the `yytext` string. `yyleng` is the number of characters matched; it does *not* include the `'\0'` (`NUL`) character that terminates the string.

Both `yytext` and `yyleng` are automatically set by the lexical analyzer before the action routine code is executed.

Matching Patterns

When the lexical analyzer runs, it tries to match the input characters against the patterns represented by the rules' REs. If more than one pattern would match the input characters, the lexical analyzer prefers the pattern that matches the *most* characters.

It's OK to have two REs that match as long as they match different numbers of characters. After all, that's how one distinguishes a pattern that's a prefix of another pattern. An example is `INTEGER_LITERAL` and `DOUBLE_LITERAL` . The string `123.456` begins with what looks like an `INTEGER_LITERAL` , `123` . However, the succeeding `.456` will also be matched by the `DOUBLE_LITERAL` pattern resulting in a longer match. Therefore `DOUBLE_LITERAL` 's pattern will be preferred. (This is know as the *maximal munch* principle—consume as many characters as possible.)

On the other hand, if multiple patterns match the *same* number of characters, the lexical analyzer prefers the pattern that occurs earliest in the *Rules Section*.

Do *not* write REs that would match the same number of characters. If you do, the action routine that gets executed will be (somewhat) arbitrarily selected, as it's based on the order of the rules. Rethink your patterns so this case does not occur.

Writing Action Routines

OK, so that's the format of a rule: a pattern RE and an action routine. It's also pretty obvious what it means to write the RE of a pattern (hey, everyone's used REs before, right?). But what about the *action routine*? What gets done in that? What *should* get done?

An action routine gets executed when its corresponding pattern RE matches the input. When the action routine is entered, the string of characters matched by the RE is available using the `char * yytext`. The length of the match (not including the trailing `NUL` terminator) is available using the `int yyleng`.

> Do *not* change `yytext` or the characters it points to or the value of `yyleng` in the action routine. There are obscure reasons one might want to do so, but they do not come up in these introductory examples.

You can do whatever processing you want in the action routine, but in general there are two cases: ① A token is recognized and must be returned and ② The characters are to be ignored.

① *Token Recognition*. In the case when a token is recognized, you must `return` a value from the action routine indicating which kind of token was seen. The `tok_INT_LIT` and `tok_ID` action routines in the `minimal1` example are of this kind.

> See *Appendix: Character Tokens* for more information on single-character tokens.

If the token kind being returned is simple, the `return` value itself identfies it. For example, recognizing the token kind *left parenthesis* is simple since all left parentheses are the same. On the other hand, recognizing the token kind *integer* is *not* simple as integers can have different values from one another.

In the case of a complex token kind such as *integer*, not only must the token kind be returned, but an indication of the *value* of this specific integer token must be made available. This is done through the `yylval` variable. In the action routine for the token kind `tok_INT_LIT`, the characters that were recognized as an integer literal (in `yytext`) are converted to an integer value (using `atoi( yytext )`) and that value is copied to the integer field `intval` of the `yylval` union.

Similar processing should be taken in the action routines of other complex token kinds: convert the characters in `yytext` to a suitable value and make that value available through the corresponding field of the `yylval` union. The definition of the `yylval` union should be updated to include whatever alternate fields are required to hold the values corresponding to those token kinds.

> The integer value of the token kind returned from the action routine must not be equal to `0`. Token kind `0` is used to indicate that the end of the token stream has been reached and there are no more tokens to retrieve.

② *Ignoring Characters*. Sometimes characters are recognized but do not form part of any token. A typical example of this would be whitespace. Most programming languages are *free form* in that the specific indentation of the code does not matter. In such languages, the only meaningful whitespace is that inside string or character literals or that used to separate otherwise ambiguous tokens.

Even though the whitespace is meaningless, the characters that comprise it must be consumed from the input character stream. After matching, though, there is no token to return, so no `return` statement should exist in the action routine. Just drop off the end of the action routine. At that point the lexical analyzer will begin trying to match the *next* set of characters.

Aside from whitespace, another reason to ignore a character is that it is *illegal*, that is, it doesn't occur in the construction of a legal token. In this case, the character should be consumed but there is no token to return.

The `minimal` examples show how to consume whitespace as well as how to report illegal characters. Study their processing of these two cases.

Another item that should be carefully considered in every action routine is the tracking of the current column number. While the generated lexical analyzer automatically counts the *line* number (in `yylineno`), the *column* position must be tracked manually.

This is done in a simple way in `minimal1` (using `yycolno`) and in a more sophisticated way in `minimal2` (using `yycolnoBegin` and `yycolnoEnd`). Study those two examples carefully so that you can make more useful error messages.

User Code Section

The *User Code Section* includes any C code that you want to be put in the lexical analyzer file `lex.yy.c`. It's useful for the body of routines that get used by action routines. The `minimal` examples are so simple we even put the `main` routine here.

Anything in the *User Code Section* just gets copied verbatim from the `.l` file to the `lex.yy.c` file.

> There is an important difference between ① code in the *User Code Section* and ② “Included C Code” in the *Definitions Section*.
>
> In the ① case, the C code is inserted in the `lex.yy.x` file *after* the code of the action routines. In the ② case, the C code is inserted in the `lex.yy.x` file *before* the code of the action routines. We therefore put declarations in the “Included C Code” in the *Definitions Section*. Actual routines are put in the *User Code Section*.

Usage Model

Given you have a valid set of patterns and action routines, how do you actually perform the scanning? The usage model is to ① set up the character stream, ② scan the tokens, and ③ tear down the character stream.

Set up the Character Stream

The lexical analyzer needs an input stream of characters. In the earliest `minimal` example `minimal1`, this stream comes from a character literal. This is particularly easy to set up. We use the `yy_scan_string()` routine to set the string that is to be scanned. Trivial!

In `minimal2` we want the lexical analyzer to take its input character stream from a file. In this case, we have to first open the file for input (using `fopen( fileName, "r" )`) and then use the `yyrestart()` routine on the obtained `FILE *`. Again, trivial!

Scan the Tokens

The lexical analyzer generated by `flex` is used by calling `yylex()`. This routine takes no arguments and returns the number of the token category that is next recognized. When there are no more tokens to return, `yylex()` returns `0`.

The easiest thing to do (at least in the introductory examples) is to use a `while` statement to process each token. Like this,

```
// Get the tokens one-by-one
int tok;
while ( ( tok = yylex() ) ) {
  switch ( tok ) {
    case Some-Token-ID:
      // Process a token of category Some-Token-ID
      break;

    case Some-Other-Token-ID:
      // Process a token of category Some-Other-Token-ID
      break;

    ...
    // Put the rest of the token category cases here
    ...

    default:
      // Here's the "unrecognized token category" case.
      break;
  }
}

// All done!  No tokens left.
```

The identifiers *`Some-Token-ID`* and *`Some-Other-Token-ID`* should be the token category names from the `enum` you defined, as described in *Definitions Section > Included C Code* above. Your `switch` statement here should have a `case` for each possible token kind as well as a `default` section just in case an error occurs. (Don't forget the `break` statement at the end of each block of processing!)

Tear Down the Character Stream

Once there are no more tokens, you have to tear down the character stream you set up. In the `maximal1` case, there's nothing to do; the characters were coming from a string literal. In the `maximal2` case, the characters were coming from a file. That file should be closed using `fclose()`.

Appendix: `flex` Regular Expressions

In the *Rules Section* `flex` uses Regular Expressions (REs) to describe the patterns of each rule. As is usual with a tool that uses REs, `flex` has its own peculiarities of what is and isn't an RE and its own syntax for expressing them. Fortunately, `flex`'s notions are fairly standard. We present an introductory summary of `flex`'s syntax for REs here, with some description. More complex RE structures (if you need them) can be found in `flex`'s documentation.

In the following summary, `w`, `x`, `y`, and `z` stand for *any* characters. `R` and `S` stand for *any* RE. `n` and `m` stand for any non-negative integers.

> Remember that in the *Rules Section* the a patterns's RE must start in first column (no whitespace before it) and the `{` of its action routine must be one the same line as this RE and separated from the RE by whitespace.

Characters

`.`

Matches *any* character *except* newline.

`w`

Matches the character `w`. (Read *Special Cases* below.)

`\w`

If `w` is `a`, `b`, `f`, `n`, `r`, `t`, or `v`, `\w` matches the C interpretation of the escape sequence `\w`. For example, `\n` matches newline, `\t` matches the tab character. If `w` is *not* one of those seven special characters, `\w` matches the explicit character `w`. For example, `\\` matches a `\` character.

Unlike some RE implementations, `flex` gives no special meaning to sequences such as `\d`, `\s`, `\w`, etc. `flex` uses *Defined Names* for this purpose.

`\0`

The `NUL` character.

`\123`

The character with *octal* value `123`, where `123` can be any octal value from `000` to `377`.

`\x12`

The character with *hexadecimal* value `12`, where `12` can be any hexadecimal value from `00` to `FF`.

`"..."`

The characters within the quotation marks `"` are matched *literally*. No character inside the marks has any special meaning. For example, `"[a-z]+"` means to literally match the six characters `[`, `a`, `-`, `z`, `]`, and `+`. No special interpretation is given to any character inside the `"` marks.

Character Classes

In general, characters classes are used to match any one of a given set of characters. The character class syntax `[ ... ]` is used to make it easier to specify the sets. Within the brackets, no character has any special meaning unless explicitly mentioned below. If an explicit `]` character is desired inside a class, it must be the *first* character listed.

`[wxyz]`

Match either of the `w`, `x`, `y`, or `z` characters. You can have as many individual characters as you want inside the `[ ]` marks and any *one* of them will be matched. Inside a character class, the `\w` construct may be used and will be properly interpreted. For example, `[ \t]` is a character class that matches the space or tab character.

`[^wxyz]`

Using a `^` as the *first* character inside a character class *negates* its meaning. In this example, this means that any character *except* `w`, `x`, `y`, or `z` will be matched. This interpretation means a newline `\n` will be matched. If you don't want a newline matched, be certain to explicitly exclude it.

If an explicit `^` character is desired inside a class, it must *not* be the first character listed.

`[0-9]`

Using a `-` inside a character class indicates a *range* of characters is to be matched. In this example any one of the characters `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9` will be matched. `flex` will let you use *any* characters in *any* order when expressing a range. Ensure that the characters you pick make sense in the order they are listed or some very unexpected results are possible.

If an explicit `-` character is desired inside a class, it must be the *first* or *last* character listed.

`[A-Z]{-}[AEIOU]`

`{-}` between two character classes performs *set difference*. In this case, the first character class matches all uppercase letters. The second character class matches all uppercase vowels. By performing this set difference, the resulting character set is all uppercase consonants.

RE Modifiers

`R*`

Match zero or more of the RE `R`.

`R+`

Match one or more of the RE `R`.

`R?`

Match zero or one of the RE `R`.

`R{n}`

Match `n` occurences of the RE `R`.

`R{n,m}`

Match `n` through `m` occurences of the RE `R`.

`R|S`

Match either the RE `R` or the RE `S`.

`( ... )`

Group the contents into a single RE. The grouping construct may be used affect how much the modifiers affect.

Match Modifiers

`^`

Match the beginning of a line. This must be the *first* character of an RE.

`$`

Match the end of a line. This must be the *last* character of an RE.

`(?i:R)`

Match the RE `R` in a case-insensitive way.

Defined Names

As described in *Definitions Section* > *Definitions* above, `flex` supports the creation of *Defined Names*. The definition of such a name goes in the *Definitions Section*. Such a name is used in a pattern's RE by enclosing the name in braces `{ ... }`. For example, the definition,

```
DIGIT        [0-9]
```

may be used in a rule thusly,

```
{DIGIT}*({DIGIT}\.|\.{DIGIT}){DIGIT}*    {
    printf( "Saw float value %f, from characters \"%s\".\n",
        atof( yytext ), yytext );
}
```

Special Cases

The quotation character `"` should be written in a character class, that is, as `["]`, unless you are trying to write an explict string of characters to match `" ... "`. This always comes up when students try to write the RE for a string literal. It's natural to want to write `"[^"\n]*"` as the RE for the pattern. Even though in the abstract that seems as if it *ought* to work, it fails in a fairly obscure way, matching only the explicit characters `[`, `^`, etc. Instead, write something like `["][^"\n]*["]`. The `"` characters are inside the character class notation and will not have their special meaning.

> You can also write `\"`, which also matches a literal `"` character just as `["]` does.

Another special case is whitespace itself. Sometimes a pattern may require whitespace characters to be recognized either at the beginning of the RE or at its end. Either way, trying to match whitespace there causes problems as `flex` forbids whitespace at the beginning of the pattern and uses whitespace at the end of the pattern to separate the pattern from the action routine. Instead, escape it (i.e., write `\` instead of just) or use a character class (i.e., `[ ]`) to hold the whitespace.

Appendix: Character Tokens

Most programming languages have a variety of, e.g., punctuation marks and operators that are single characters. For example, C has `+`, `-`, `*`, and `/` for the four fundamental arithmetic operations. Normally, we would define a separate token ID for each of these, perhaps `tok_PLUS`, `tok_MINUS`, `tok_MULTIPLY`, and `tok_DIVIDE`. While possible, this is pretty tedious. (And, as you'll see when we begin with `bison`, it obfuscates the grammar rules.)

`flex` provides a mechanism for avoiding this complexity. The `yylex()` routine returns an integer value indicating the ID of the scanned token (or `0` at the end of the token stream). The integer values `1` through `255` are *reserved* so that they can be used to indicate a single `ASCII` character. That is, the token ID value `40` is used to represent the *left parenthesis* token because `40` is the `ASCII` value of the `(` character. For example, the following rule is quite convenient,

```
[(]    { return '('; }
```

This rule recognizes a left parenthesis and returns `40` as the token ID. We get away with this in C because a character literal, `'('` here, is an `int` with the value of the `ASCII` code of the character. Pretty neat, eh?

We can extend this concept to recognize *all* of the single character tokens of our language by writing something like this,

```
[-+*/()[\]{}<>]    { return yytext[0]; }
```

This pattern uses a character class to match one instance of any of the characters given in the class. Here we've included not only the four fundamental arithmetic operators but also the fours sets of bracketing characters.

> The `-` character has to be first in the character class so that it's recognized as a literal `-` character and not interpreted as indicating a range of characters. Similarly, we have to write `\]` to ensure that it does not close the character class.

In the action routine, the `ASCII` value of the matched character is returned as the token ID. We retrieve it as the first character in `yytext`. Really, really neat, eh?

Appendix: Distinguishing Keywords from IDs

Most programming languages have the concept of an *identifier*, a name created by the user representing some object in the program. These languages usually also have a number of *keywords* (also known as *reserved words*), names that have the same form as identifiers but that are reserved by the language itself.

For example, C has (since C11) these keywords,

auto	float	signed	_Alignas
break	for	sizeof	_Alignof
case	goto	static	_Atomic
char	if	struct	_Bool
const	inline	switch	_Complex
continue	int	typedef	_Generic
default	long	union	_Imaginary
do	register	unsigned	_Noreturn
double	restrict	void	_Static_assert
else	return	volatile	_Thread_local
enum	short	while	
extern			

The form of these keywords is the same as identifiers so there will be a clash between a rule to recognize one of the keywords and the general rule to recognize an identifier. For example, in the *Definitions Section*, we could write,

```
   //---- DEFINITIONS --------
%{
enum {
  ...
  tok_AUTO,
  tok_BREAK,
  tok_CASE,
  ...
  tok_INLINE,
  ...
  tok__STATIC_ASSERT,
  tok__THREAD_LOCAL,
  ...
};

%}
```

We would then write a *Rules Section* such as this,

```
%% //---- RULES --------------

auto                    { return tok_AUTO; }
break                   { return tok_BREAK; }
case                    { return tok_CASE; }
...
inline                  { return tok_INLINE; }
...
_Static_assert          { return tok__STATIC_ASSERT; }
_Thread_local           { return tok__THREAD_LOCAL; }

[a-zA-Z_][a-zA-Z_0-9]*  { return tok_ID; }
```

(The `...` spots indicate where lines have been omitted.)

If the character stream has for example as its next letters `inline`, two patterns will match: the one for `tok_INLINE` and the one for `tok_ID`. Because the `tok_INLINE` rule occurs earlier than the rule for `tok_ID`, its action routine will be executed and the characters will be scanned as the reserved word `tok_INLINE` instead of the general `tok_ID`.

As mentioned above, having two patterns that match the same characters is bad style. We therefore would rather have a way to process keywords and identifiers in a unified way. In particular, in the *Definitions Section*, where we created `enum` constants for the token numbers for the keywords, we put a forward declaration for the `isKeyword()` function, which we define in the *User Code Section*.

```
   //---- DEFINITIONS --------
%{
enum {
  ...
  tok_AUTO,
  tok_BREAK,
  tok_CASE,
  ...
  tok_INLINE,
  ...
  tok__STATIC_ASSERT,
  tok__THREAD_LOCAL,
  ...
};

extern int isKeyword( const char *str );
%}
```

In the *Rules Section*, we have only a single rule that is used to handle both keywords and IDs. Notice that in the action routine for this rule, we check whether the matched text (in `yytext`) is a keyword using the `isKeyword()` function. If it is, we return the indicated token number. If the matched text is not a keyword, we return `tok_ID` instead.

```
%% //---- RULES --------------

[a-zA-Z_][a-zA-Z_0-9]*    {
  int kw = isKeyword( yytext );

  if ( kw != 0 ) {
    // It's in the keyword list so return its token number.
    return kw;
  } else {
    // It's not in the keyword list so return tok_ID.
    return tok_ID;
  }
}
```

In the *User Code Section*, we define a structure type `Keyword` and an array of known keywords `keywords[]`. The `isKeyword()` function takes a pointer to the matched characters as input and looks through the `keywords[]` array to see if they line up with a known keyword. If so, the corresponding token number is returned, otherwise `0`, indicating that the matched characters are *not* a keyword.

```
%% //---- USER CODE ----------

typedef struct {
  char *kw;
  int   tok;
} Keyword;

Keyword keywords[] = {
  { "auto",             tok_AUTO; },
  { "break",            tok_BREAK; },
  { "case",             tok_CASE; },
  ...
  { "inline",           tok_INLINE },
  ...
  { "_Static_assert",   tok__STATIC_ASSERT; },
  { "_Thread_local",    tok__THREAD_LOCAL; },
};

#define NUM_KEYWORDS ( sizeof(keywords) / sizeof(Keyword) )

int isKeyword( const char *str )
{
  for ( size_t i=0; i < NUM_KEYWORDS; i++ ) {
    if ( strcmp( str, keywords[i].kw ) == 0 ) {
      return keywords[i].tok;
    }
  }

  return 0;
}
```

By using this technique to separate keywords from identifiers, we simplify the *Rules Section* and avoid ambiguous matching. It's tidier.