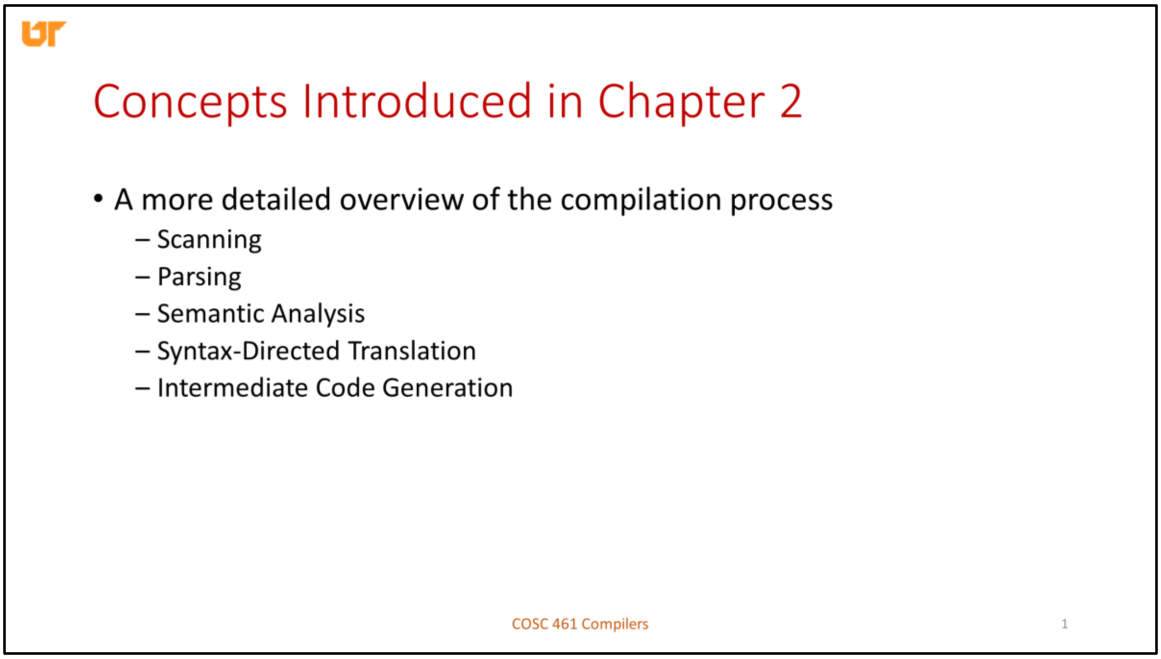


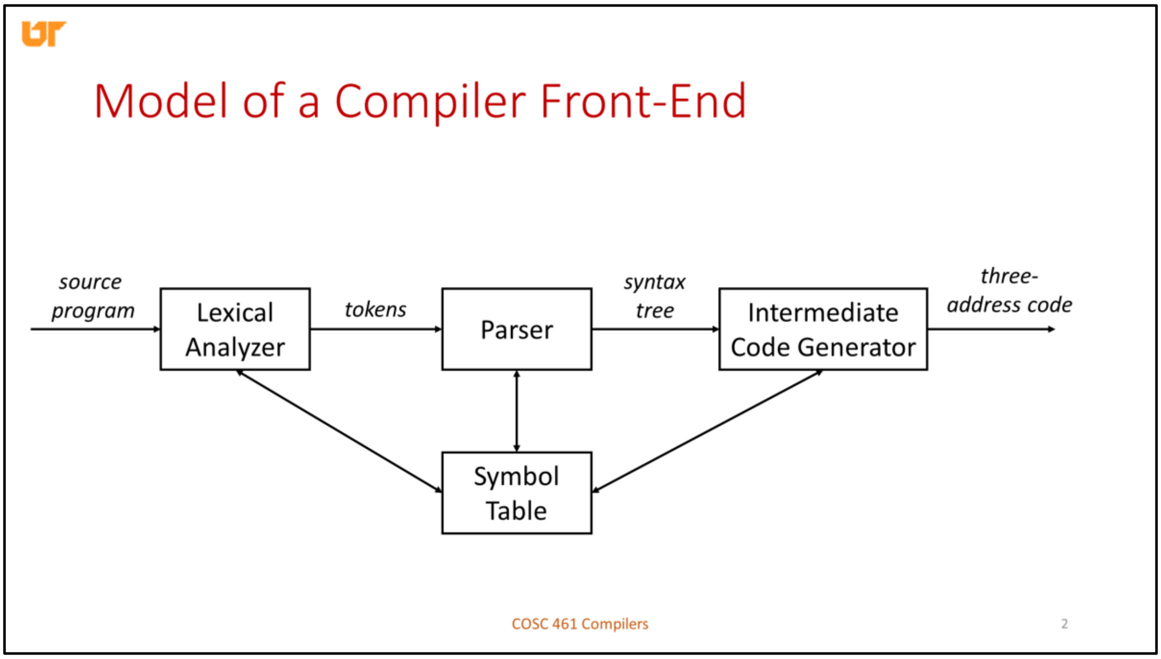
Concepts Introduced in Chapter 2

- A more detailed overview of the compilation process
  - Scanning
  - Parsing
  - Semantic Analysis
  - Syntax-Directed Translation
  - Intermediate Code Generation

Chapter 2 gives a more detailed introduction to the compilation process during the front end of the compiler (up to intermediate code generation) and introduces in more detail many of the concepts that will be later covered in the course.

So in this chapter, we'll cover …

Do not worry if some of the concepts are unclear at this time.



Model of a Compiler Front-End

source program

Lexical Analyzer

tokens

Parser

syntax tree

Intermediate Code Generator

three-address code

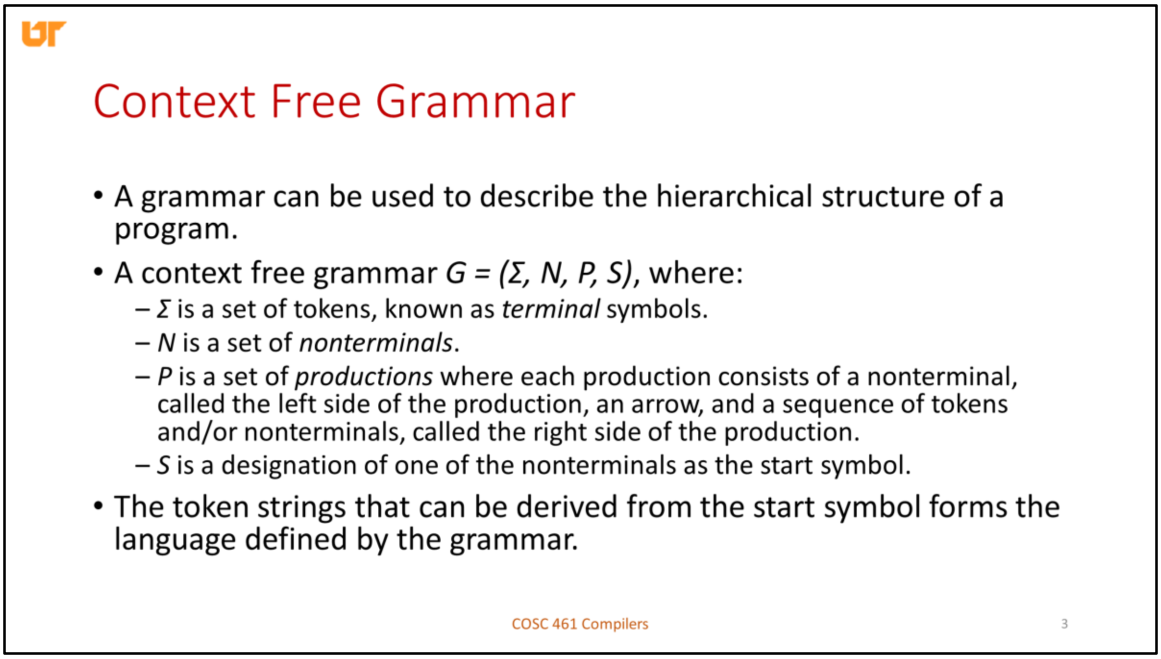
Symbol Table

COSC 461 Compilers 2

Here we have our model of the compiler front end.

The source program is fed to a lexical analyzer, which uses rules to break up the raw input stream into a sequence of tokens.
The tokens are then read by a syntax analyzer or “parser” that uses a formal syntax defined by the grammar representing the language to guide the translation.
The parser produces a syntax tree, that is further translated into some intermediate-language format, such as three-address code. Some parsers might combine a syntax analyzer and an intermediate code generator into one component.

All of these components interact with the symbol table, and might place entries into or take entries out of the symbol table.



Context Free Grammar

- A grammar can be used to describe the hierarchical structure of a program.
- A context free grammar *G = (Σ, N, P, S)*, where:
  - *Σ* is a set of tokens, known as *terminal* symbols.
  - *N* is a set of *nonterminals*.
  - *P* is a set of *productions* where each production consists of a nonterminal, called the left side of the production, an arrow, and a sequence of tokens and/or nonterminals, called the right side of the production.
  - *S* is a designation of one of the nonterminals as the start symbol.
- The token strings that can be derived from the start symbol forms the language defined by the grammar.

First, I'd like to talk about how we formally define programming languages.

Context free grammars, or what I'll simply call "grammars", are used to specify the syntax of a programming language. Grammars naturally describe the hierarchical structure of most programming language constructs.

Grammars are a precise way to define a programming language's structure that is easy to understand. Describing your language as a grammar also makes it possible to use tools that automatically produce a parser (or syntax analyzer) for your language.

So, a grammar can be described as a quadruple of sigma, N, P, and S, where:

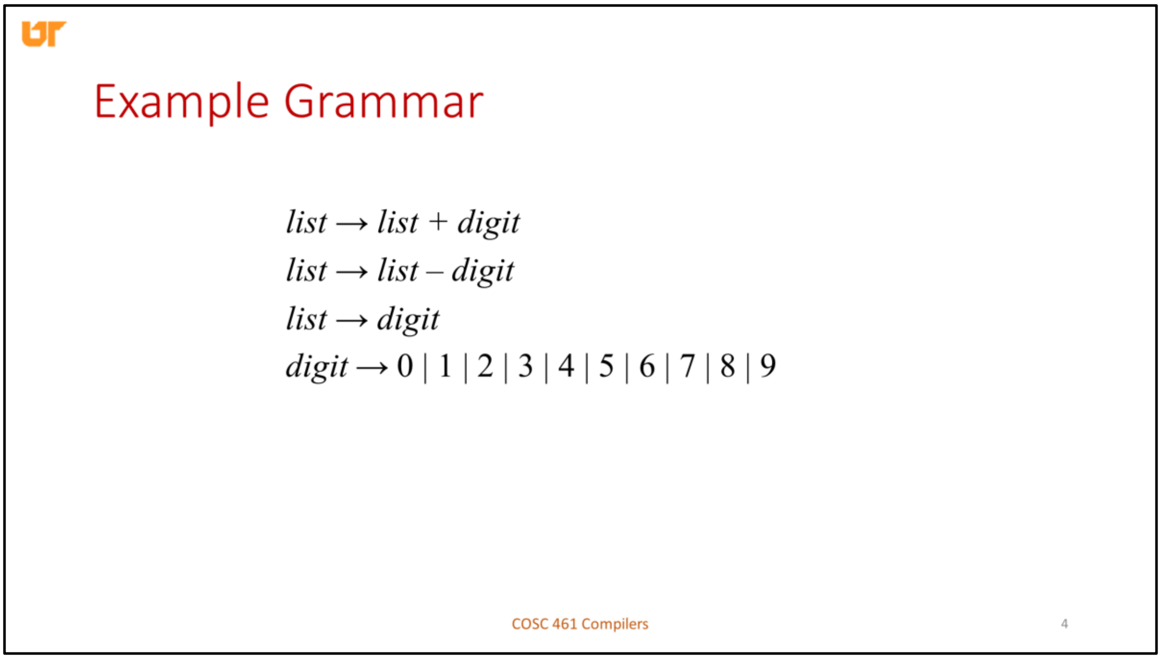
Sigma is a set of tokens, known as terminal symbols. These are the elementary symbols in your language. From now on, I'll use the words 'tokens' and 'terminals' interchangeably to describe these symbols.
N is a set of nonterminal symbols. Non-terminal symbols are always made up of a string of terminal symbols.
P is a set of productions, where each production consists of a non-terminal symbol on the left hand side of the production, an arrow, and a sequence of tokens and non-

terminals on the right hand side of the production. Productions can be thought of as rules in your grammar.
And S is a designation of one of the non-terminals as the start symbol.

So, non-terminals will be on the LHS and possibly on the RHS of the productions.
And terminals will only be on the RHS of a production.

The token strings that can be derived from the start symbol form the language defined by the grammar.

So, note that the language accepted by the grammar will only consist of tokens (terminals).



Example Grammar

$$\begin{aligned} &list \rightarrow list + digit \\ &list \rightarrow list - digit \\ &list \rightarrow digit \\ &digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

COSC 461 Compilers 4

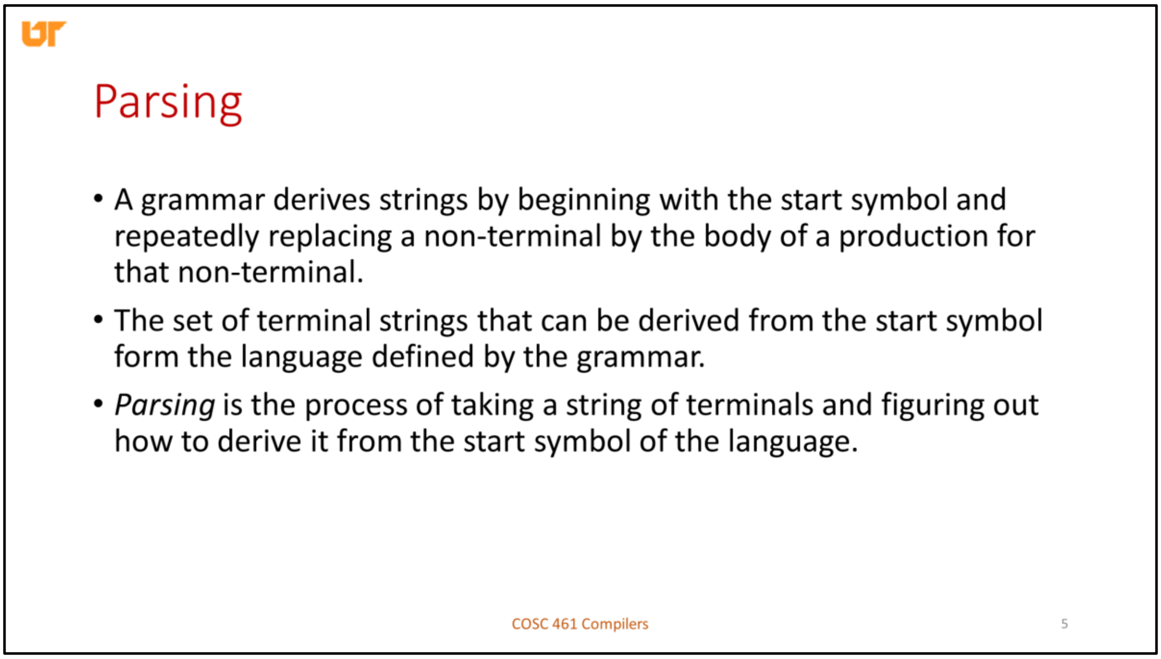
Here is an example grammar. Italicized words are non-terminals, and anything not italicized can be considered a terminal symbol. The '|' character should be read as 'or'. So, this production says digit goes to 0 or 1 or 2 …

Q. Can someone tell me what this grammar is describing?
A. This grammar describes lists of digits with + or - operators between them.

Is this in the language?
9-5+2 (yes)

list → list + digit → list – digit + digit → digit – digit + digit → 9 – digit + digit → 9 – 5 + digit → 9 – 5 + 2

Or this:
96+2 (no)
(no way to concatenate digits)



Parsing

- A grammar derives strings by beginning with the start symbol and repeatedly replacing a non-terminal by the body of a production for that non-terminal.
- The set of terminal strings that can be derived from the start symbol form the language defined by the grammar.
- *Parsing* is the process of taking a string of terminals and figuring out how to derive it from the start symbol of the language.

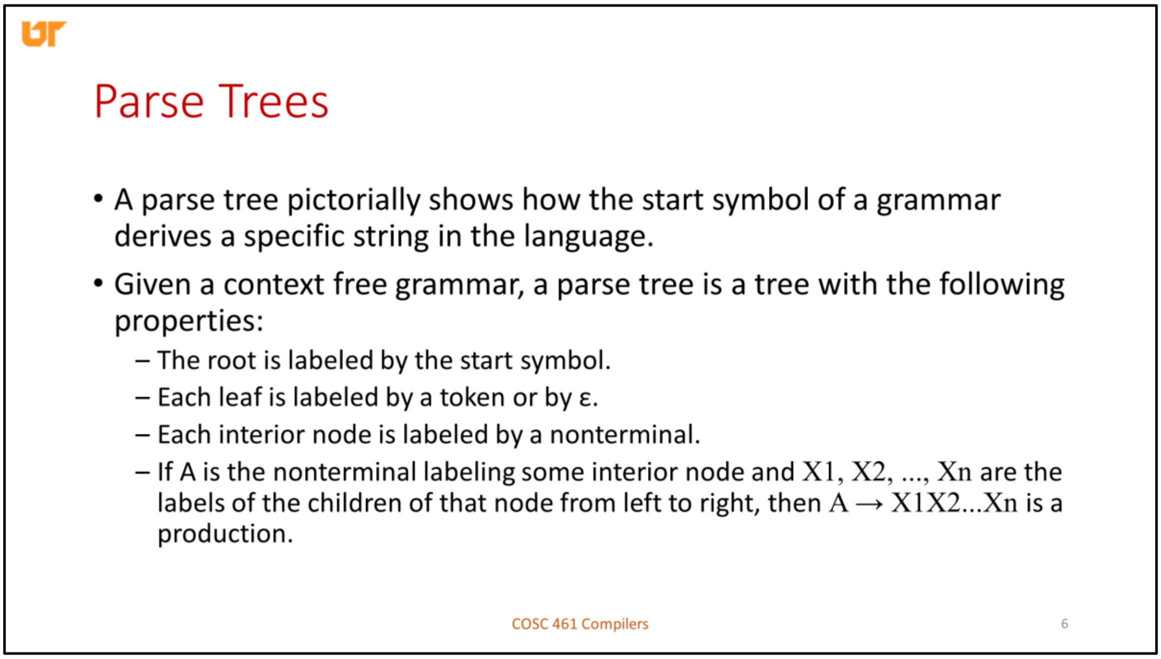
OK, so a grammar derives strings in the language by beginning with the start symbol and repeatedly replacing a non-terminal by the body of a production for that non-terminal.

Some non-terminals can have more than one right-hand side, so, in that case, we can select any one production. If our initial selection does not derive the desired terminal string, then we may have to backtrack and select another right-hand side for that non-terminal.

The set of terminal strings that can be derived from the start symbol form the language defined by the grammar. Alternatively, if the desired string cannot be derived then the string does not belong to the grammar.

Parsing is the process of taking a string of terminals and figuring out how to derive it from the start symbol of the language.

Parsing is one of the most fundamental problems in compilation. For example, we can write a grammar that accepts all valid programs in some programming language, for instance, C. Then, to check if some string is a valid C program, we just parse the string to find if our grammar can produce the program. The parsing algorithm can also report syntax errors preventing the string from being recognized as a string in our grammar.



Parse Trees

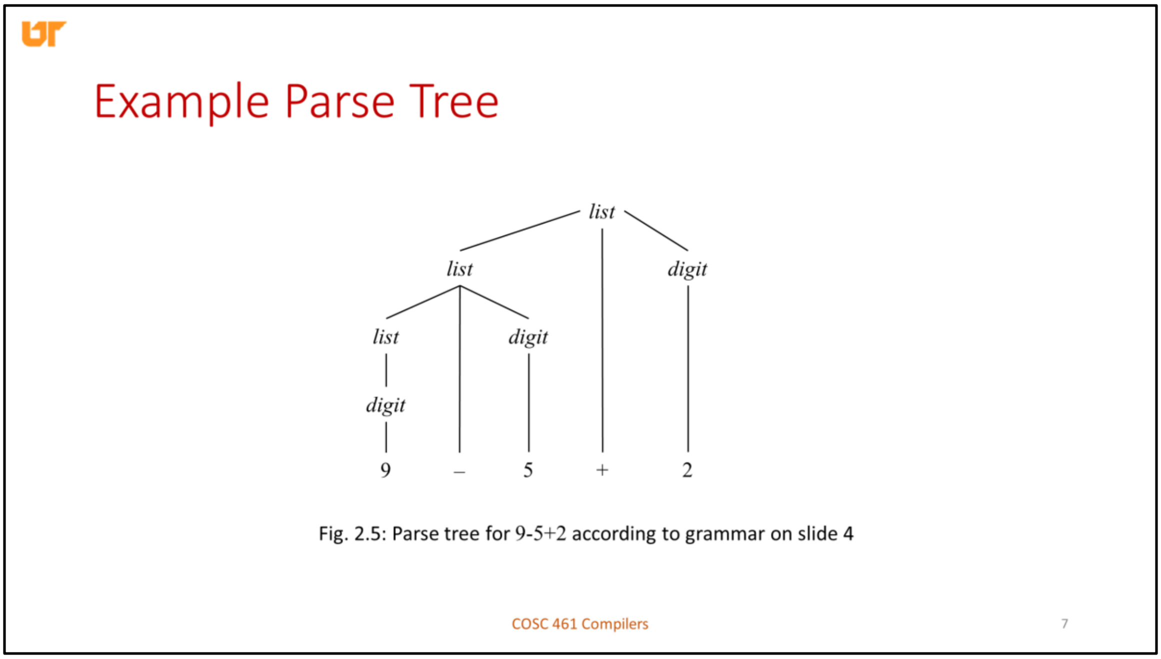
- A parse tree pictorially shows how the start symbol of a grammar derives a specific string in the language.
- Given a context free grammar, a parse tree is a tree with the following properties:
  - The root is labeled by the start symbol.
  - Each leaf is labeled by a token or by ε.
  - Each interior node is labeled by a nonterminal.
  - If A is the nonterminal labeling some interior node and $X1, X2, ..., Xn$ are the labels of the children of that node from left to right, then $A \rightarrow X1X2...Xn$ is a production.

A parse tree pictorially …

Parse trees are nice for showing how a particular string is in the language defined by a grammar.

Given a context free grammar, a parse tree is a tree with the following properties: …

(epsilon is the empty string – it is a string of zero terminal symbols)



Example Parse Tree

Fig. 2.5: Parse tree for 9-5+2 according to grammar on slide 4

Here is an example parse tree for parsing the list ‘9-5+2’ according to the grammar on slide 4.

We start with the non-terminal list and follow productions in the grammar to complete the parse tree.

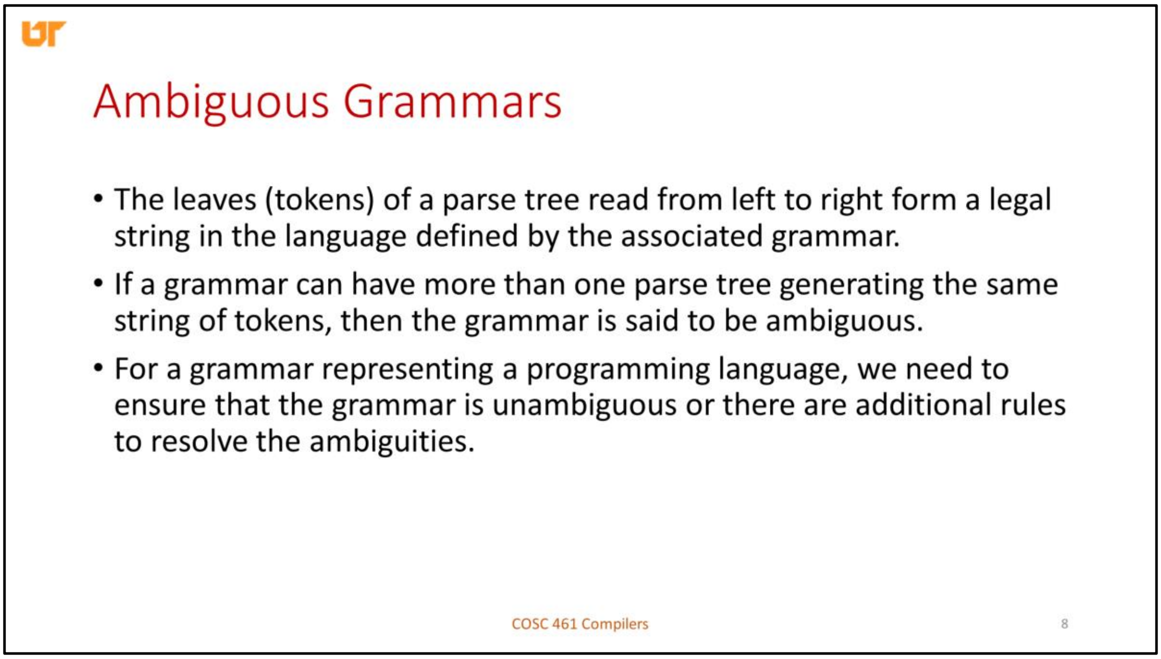
Note that the interior nodes are other non-terminals, and the leaves are terminals.

Each interior node and its immediate children correspond to a single production.

So, this figure shows that if we want to derive the string 9-5+2, one way we could do that is with this derivation. In our derivation we only apply a single production at each step:

list → list + digit → list – digit + digit → digit – digit + digit → 9 – digit + digit → 9 – 5 + digit → 9 – 5 + 2

So, from left to right, the leaves of a parse tree form a string that’s in our language.



Ambiguous Grammars

- The leaves (tokens) of a parse tree read from left to right form a legal string in the language defined by the associated grammar.
- If a grammar can have more than one parse tree generating the same string of tokens, then the grammar is said to be ambiguous.
- For a grammar representing a programming language, we need to ensure that the grammar is unambiguous or there are additional rules to resolve the ambiguities.

So, a parser can tell you the structure of a string according to a grammar. One complication, however, is that we have to deal with the problem of ambiguous grammars.

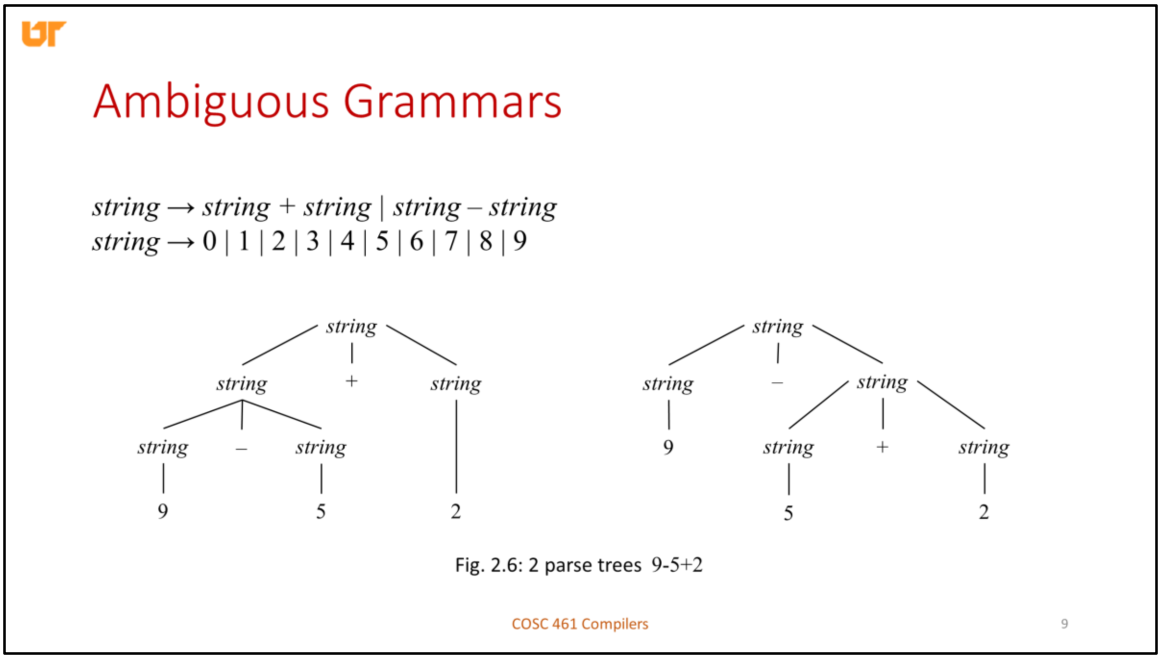
As I mentioned, the leaves (or tokens) of a parse tree read from left to right form a legal string in the language defined by the associated grammar.

If a grammar can have more than one parse tree generating the same string of tokens, then the grammar is said to be ambiguous.

This presents a problem for implementing parsers since there may be multiple ways to interpret the same string of tokens.

Thus, if we can construct multiple parse trees that generally means that the input string can have more than one meaning. For programming languages, we need to ensure that the grammar is unambiguous or that we have additional rules to resolve the ambiguities.

It has been proven that the general question of whether or not a grammar is ambiguous is undecidable. No algorithm exists to determine the ambiguity of a grammar. However, we can write our grammars in certain ways to guarantee that they are unambiguous. We’ll go over that later in the course.



Ambiguous Grammars

$$string \rightarrow string + string \mid string - string$$
$$string \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Fig. 2.6: 2 parse trees 9-5+2

Here’s an example of an ambiguous grammar. These are two possible parse trees for the input string 9-5+2 when we parse it according to the grammar shown on the slide.

Notice that the leaves of the two parse trees from left to right are the same.

But the trees correspond to two different ways of parenthesizing the expression: (9-5)+2 on the left and 9-(5+2) on the right

Notice that this could also lead to different result. If we do the lower operations in the tree first, then the left tree result would be 6, and the right tree result would be 2.

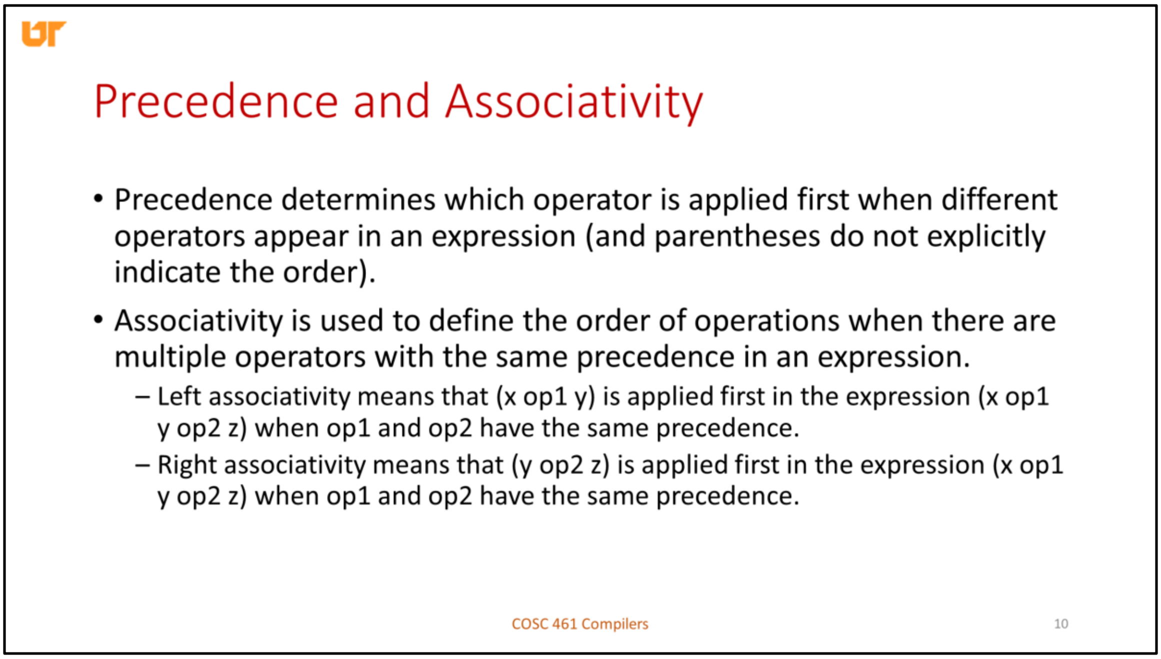
Note that, the previous grammar in Example 2.1 (Figure 2.5) was unambiguous. If we try to derive 9-5+2 in another way. Instead of starting with:

list -> list + digit

We could start with:

list -> list – digit

But then we cannot proceed. And we find that we have only one way to parse that string, so, we can only have one parse tree.



Precedence and Associativity

- Precedence determines which operator is applied first when different operators appear in an expression (and parentheses do not explicitly indicate the order).
- Associativity is used to define the order of operations when there are multiple operators with the same precedence in an expression.
  - Left associativity means that (x op1 y) is applied first in the expression (x op1 y op2 z) when op1 and op2 have the same precedence.
  - Right associativity means that (y op2 z) is applied first in the expression (x op1 y op2 z) when op1 and op2 have the same precedence.

One way to deal with ambiguity in grammars is to introduce rules for precedence and associativity.

Precedence determines which operator is applied first when different operators appear in an expression (and parentheses do not explicitly indicate the order).

For instance, * usually has higher precedence than +.
[Describe the order of operations of (9+5*2) and (9*5+2).]

When writing a parser, precedence is established by defining the operators with higher precedence to match last.

Associativity is used to define the order of operations when there are multiple operators with the same precedence in an expression.
[Describe left and right associativity from the slide]

Most arithmetic operators have left associativity (e.g. +, -, *, /).

[+ associates to the left since an operand with '+' sign on both sides of it belongs to the operator on its left.]

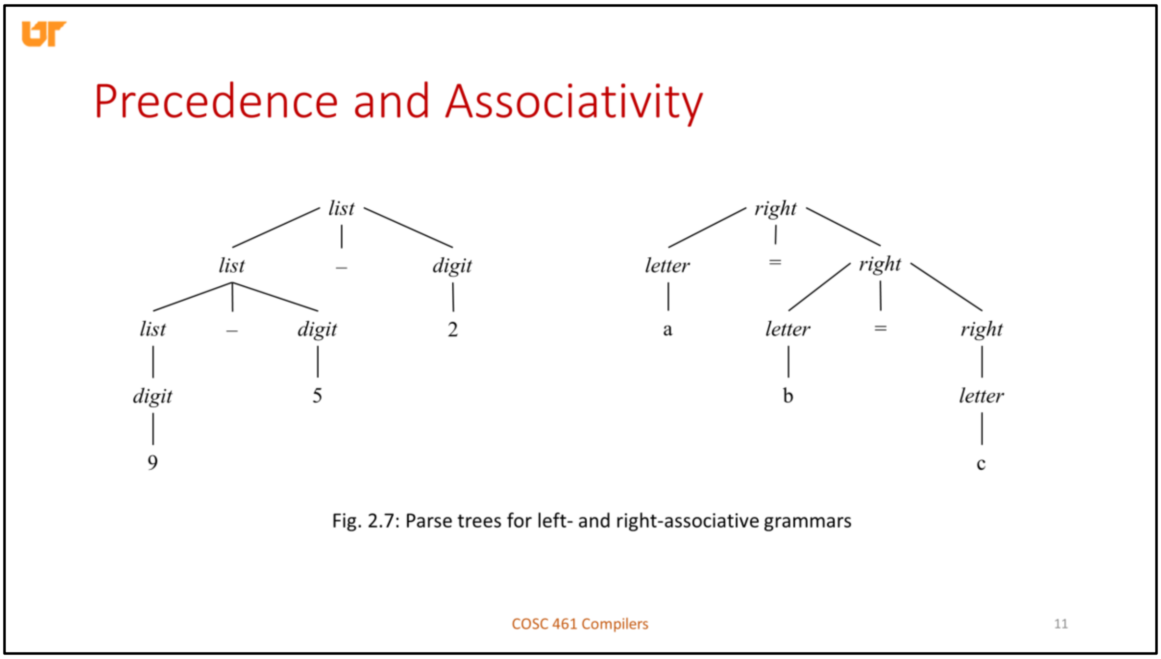
Q. Can anyone tell me an operator that has right associativity in the C programming language?

A. Not very many, but exponentials and assignment (the = operator).

Left associativity is implemented in grammars by defining productions that are left recursive.

Right associativity is implemented in grammars by defining productions that are right recursive.

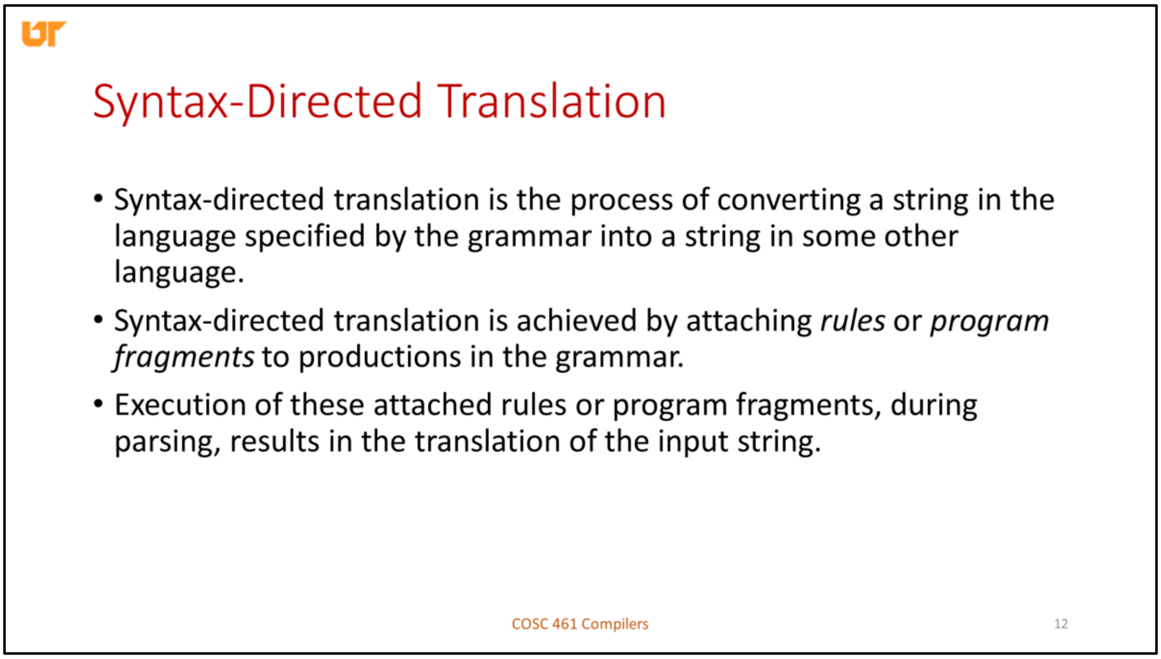
We will see examples of this in Chapter 4.



Precedence and Associativity

Fig. 2.7: Parse trees for left- and right-associative grammars

Here is an example of a parse tree for an operator that has left associativity (i.e. the –), and a parse tree for an operator that has right associativity (the ‘=‘ for assignment). Notice that the parse tree for the left associative operator grow down to the left while the tree for the right associative operators (e.g. =) grows downward to the right.



Syntax-Directed Translation

- Syntax-directed translation is the process of converting a string in the language specified by the grammar into a string in some other language.
- Syntax-directed translation is achieved by attaching *rules* or *program fragments* to productions in the grammar.
- Execution of these attached rules or program fragments, during parsing, results in the translation of the input string.

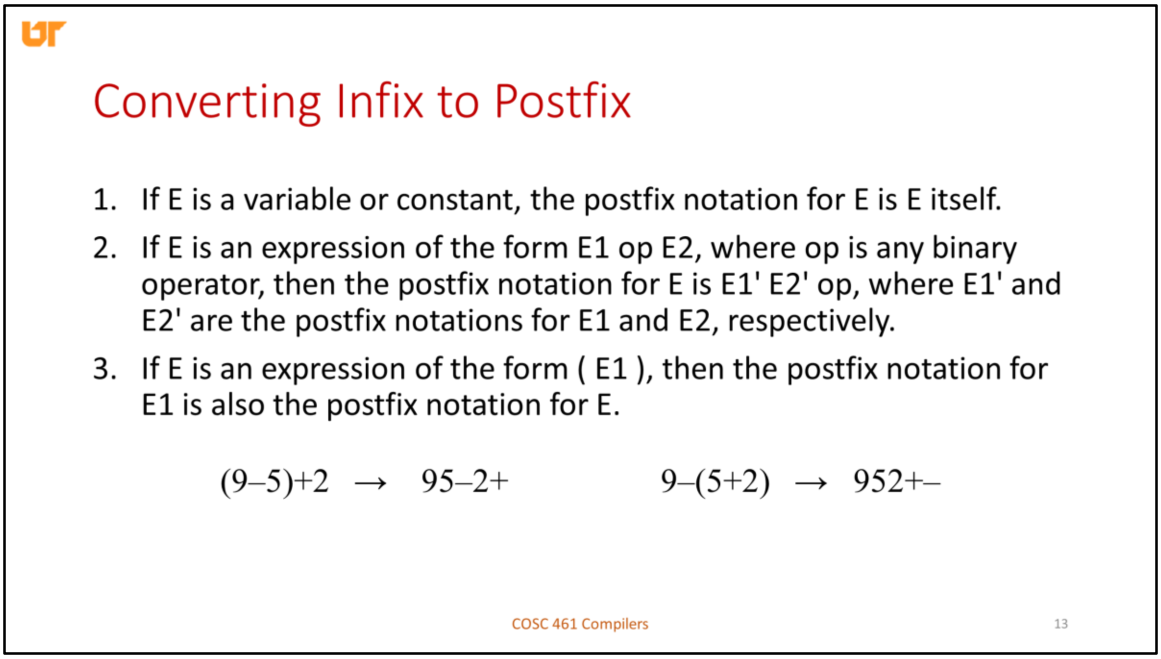
Up until now, we have seen how we can use grammars to achieve parsing to generate the set of strings in the language specified by the grammar, or even to test if the given input string belongs to the language specified by the grammar.

We can also use parsing to convert the strings in the input language to strings in some other language. For example we can use a grammar that accepts all valid C programs, and converts them into intermediate code or assembly code, or to any other language during the parsing process.

This is called syntax-directed translation.

Syntax-directed translation is achieved by attaching *rules* or *program fragments* to productions in the grammar.

Execution of these attached rules or program fragments, during parsing, results in the translation of the input string.



Converting Infix to Postfix

1. If E is a variable or constant, the postfix notation for E is E itself.
2. If E is an expression of the form E1 op E2, where op is any binary operator, then the postfix notation for E is E1' E2' op, where E1' and E2' are the postfix notations for E1 and E2, respectively.
3. If E is an expression of the form (E1), then the postfix notation for E1 is also the postfix notation for E.

(9–5)+2 → 95–2+ 9–(5+2) → 952+–

COSC 461 Compilers 13

I want to discuss a slightly more complex example to show how a grammar can be used to accomplish a translation. Specifically, we’re going to examine how we could translate infix notation to postfix.

Q. Can someone describe what infix notation is?

A. Infix expressions have the operators between the operands – this is how we normally write arithmetic expressions.

Q. So what is postfix?

A. Postfix expressions have the operators after the operands.

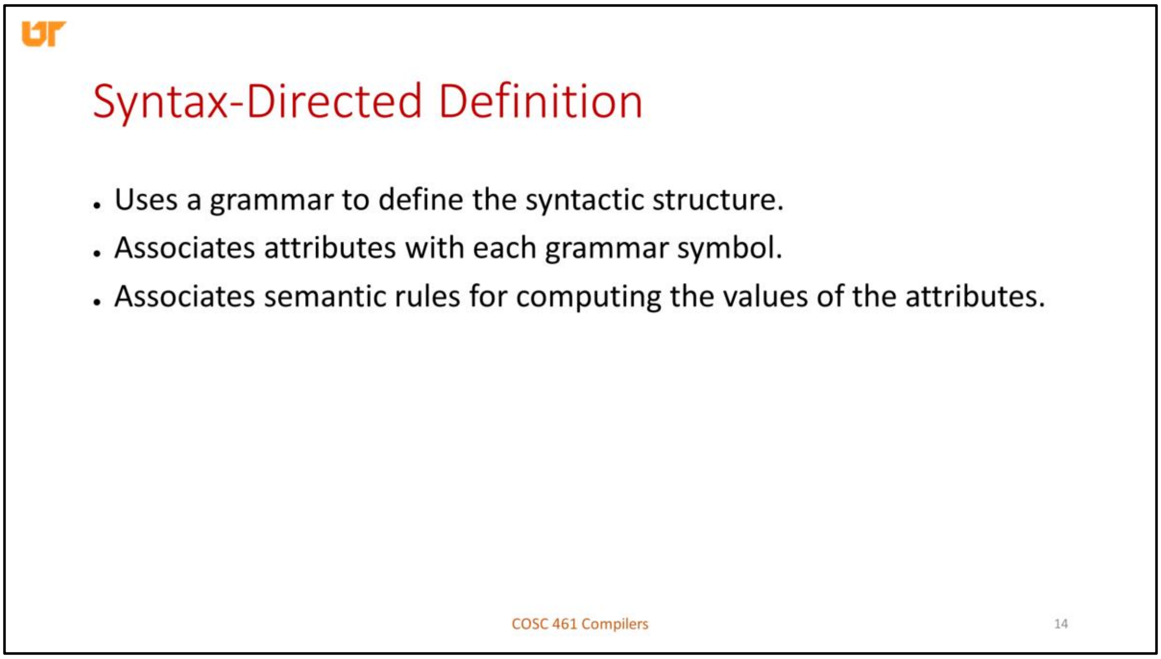
There is no need for parentheses in postfix expressions since these expressions have an unambiguous evaluation.
[Describe how to evaluate a postfix expression using a stack]

This slide describes rules we could use to translate an infix notation expression to postfix.

Read the slide – and go over the two examples

(9-5)+2 ->
Do (9-5) first -> (95-) -> 95-
Then, since we know postfix for the first part, apply rule 2 to the whole expression:
(9-5)+2 -> 95-2+

9-(5+2) ->
Do (5+2) first -> (52+) -> 52+
Now, apply rule 2:
9-(5+2) -> 952+-



Syntax-Directed Definition

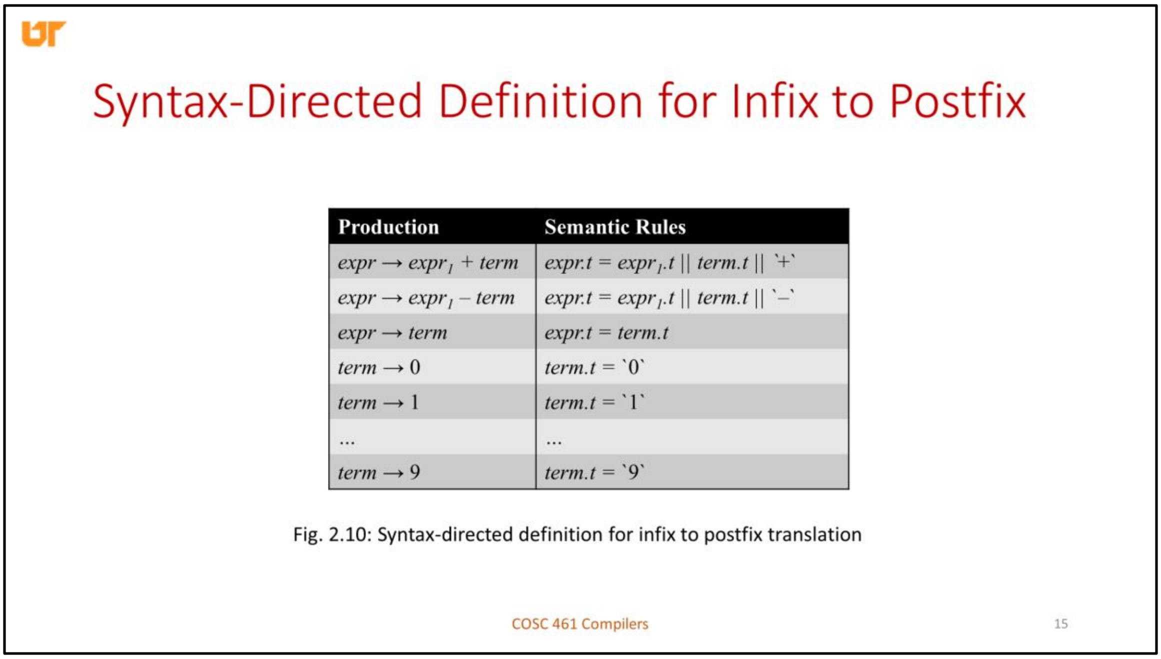
- Uses a grammar to define the syntactic structure.
- Associates attributes with each grammar symbol.
- Associates semantic rules for computing the values of the attributes.

The idea of attaching rules to grammar productions to do a translation is summarized in the syntax-directed definition.

We use a grammar to define the syntactic structure of our language.

Then, we associate attributes with each grammar symbol.

And we associate semantic rules for computing the values of these attributes. We'll go through this with an example.



Syntax-Directed Definition for Infix to Postfix

Production	Semantic Rules
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|$ \`+\`
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \,\|\|\, term.t \,\|\|$ \`−\`
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t =$ \`0\`
$term \rightarrow 1$	$term.t =$ \`1\`
…	…
$term \rightarrow 9$	$term.t =$ \`9\`

Fig. 2.10: Syntax-directed definition for infix to postfix translation

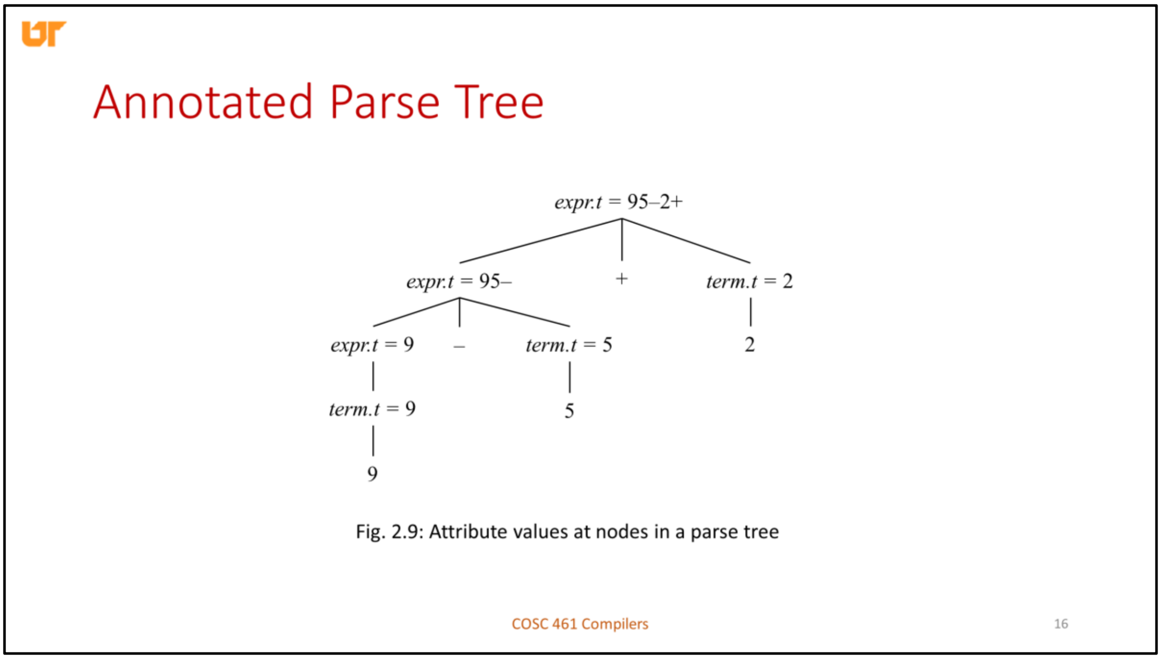
This table describes a syntax-directed definition for doing the infix to postfix translation.

The productions of the grammar are on the left, and the rules for generating postfix notation are on the right. In this table, the '||' is a string concatentation operation.

Each symbol has a string-valued attribute (t) that represents the postfix notation for the expression.

Remember that the postfix of a digit is just the digit itself. So, the semantic rule *term.t* = 9 defines the postfix form of *term* to be 9 whenever the production *term* -> 9 appears in the parse tree.

Similarly, whenever the production *expr* -> $expr_1$ + *term* is seen in the parse tree, then we have a semantic rule that constructs the postfix of *expr* as the postfix of $expr_1$ concatenated with the postfix of *term* concatenated with the '+' symbol. This rule is a formalization of rule 2.



Annotated Parse Tree

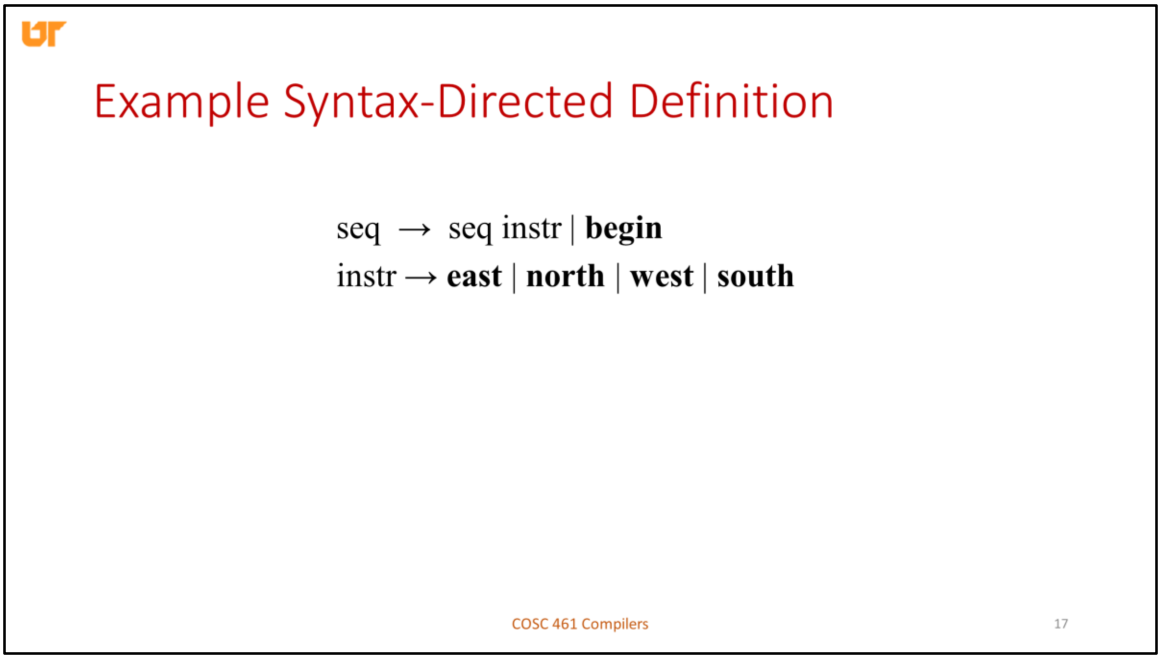
Fig. 2.9: Attribute values at nodes in a parse tree

Here is a parse tree annotated with the values of each attribute. Constructing this tree is straightforward. We first construct the (unannotated) parse tree for the input string. Then, we apply the semantic rules from the syntax directed definition at each node.

Write up the grammar.
Draw up the parse tree.
And step through the example.

In this case, all of the attributes are synthesized, which means they can be determined from the attribute values of their children.

It is desirable to have parse trees with synthesized attributes because these attributes can always be evaluated by doing a simple depth-first order evaluation of the parse tree.



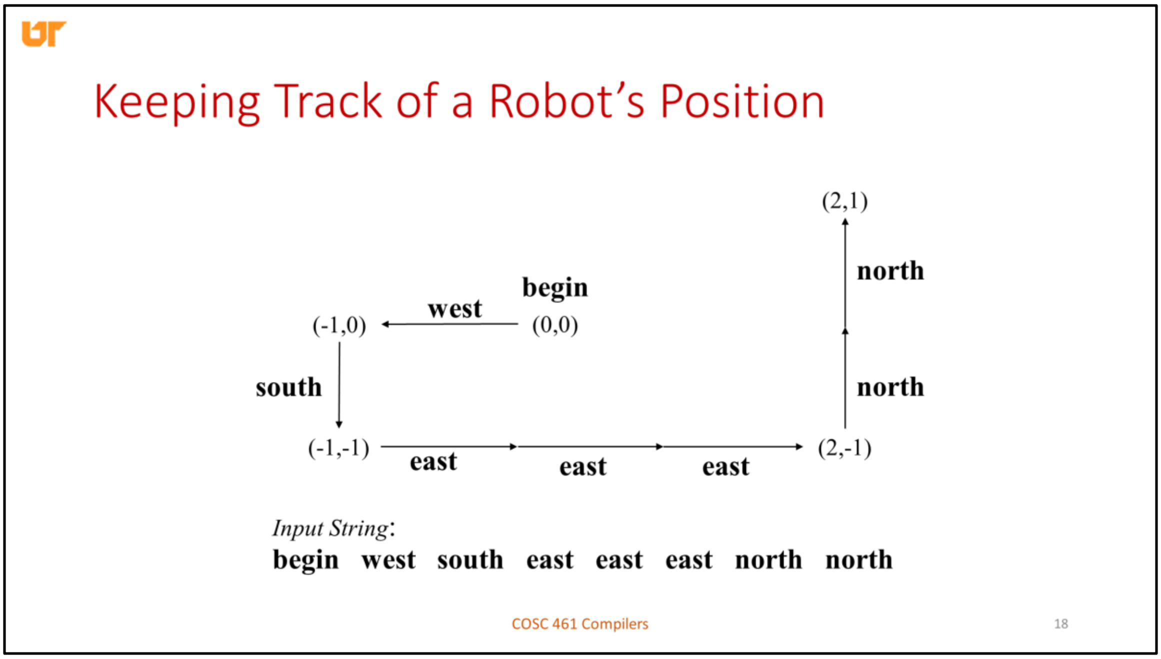
Example Syntax-Directed Definition

seq → seq instr | **begin**
instr → **east** | **north** | **west** | **south**

Now let’s discuss another example of a syntax-directed definition.

This grammar represents a robot's moves. Read it …

For this example, we’re going to use this grammar to do a translation that converts the robot’s moves into co-ordinate positions.



Keeping Track of a Robot’s Position

Input String:
begin west south east east east north north

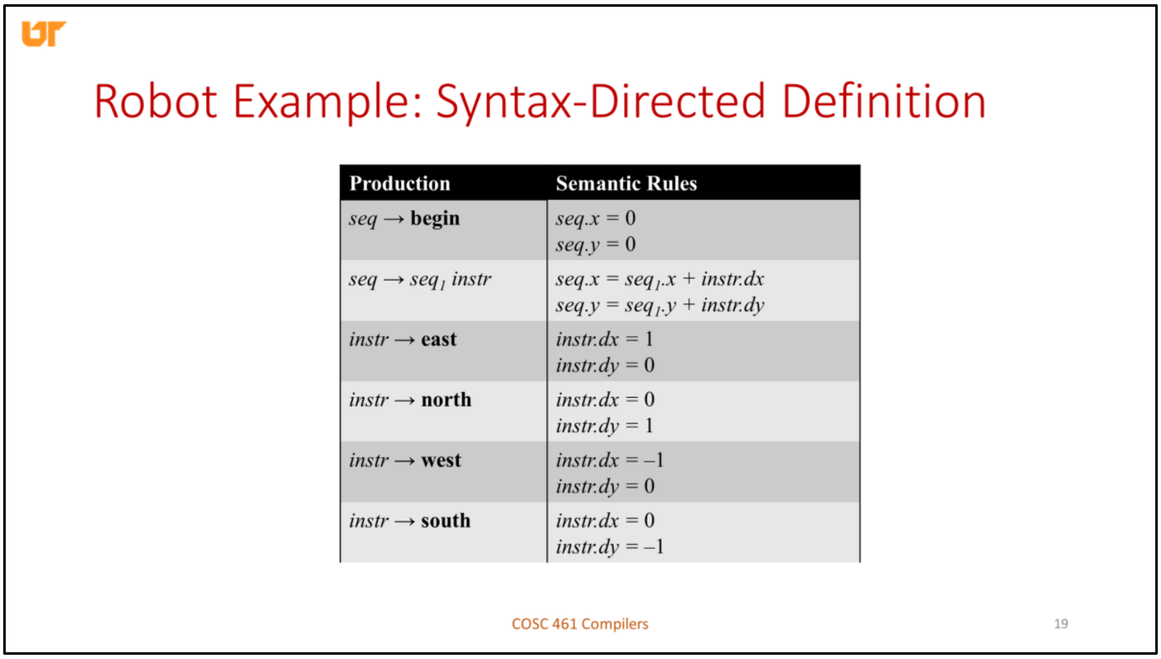
COSC 461 Compilers 18

This figure shows how we wish to update the robot's position based on moves that are taken.

So, if this is our input string, this shows the path that would be taken.

The x or y coordinate is updated depending on what instruction is encountered.

When we go west, …



Robot Example: Syntax-Directed Definition

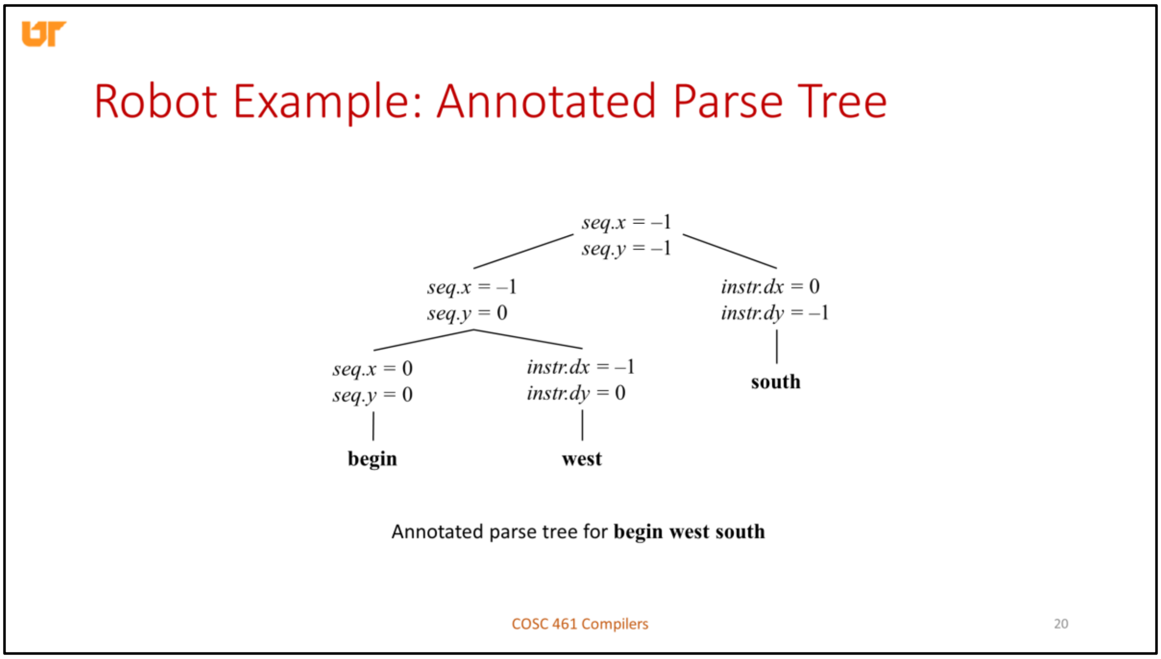
Production	Semantic Rules
$seq \rightarrow$ **begin**	$seq.x = 0$ $seq.y = 0$
$seq \rightarrow seq_1\ instr$	$seq.x = seq_1.x + instr.dx$ $seq.y = seq_1.y + instr.dy$
$instr \rightarrow$ **east**	$instr.dx = 1$ $instr.dy = 0$
$instr \rightarrow$ **north**	$instr.dx = 0$ $instr.dy = 1$
$instr \rightarrow$ **west**	$instr.dx = -1$ $instr.dy = 0$
$instr \rightarrow$ **south**	$instr.dx = 0$ $instr.dy = -1$

Here is the syntax-directed definition that formalizes our robot example.

Just like with the postfix example, each time a particular production is taken, we can update the attribute value associated with the non-terminal on the left side of the production.

Multiple moves are added together by updating the *x* and *y* attributes of the *seq* nonterminal.

We will see later that the *begin* production is actually taken first.



Robot Example: Annotated Parse Tree

Annotated parse tree for **begin west south**

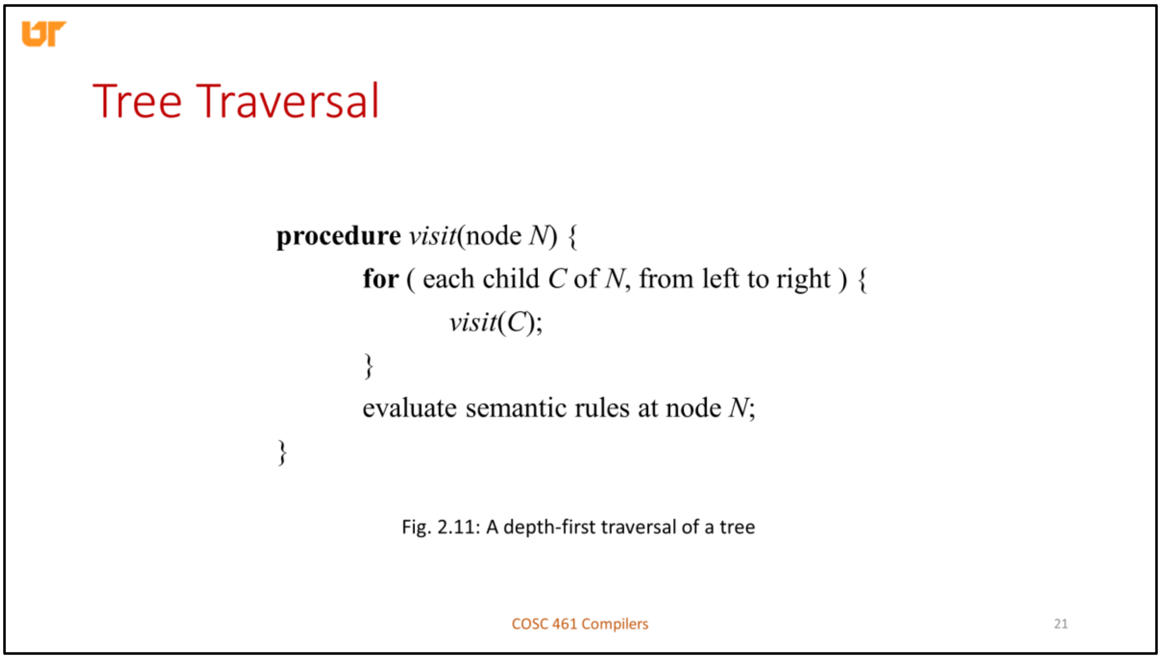
Here is the annotated parse tree for parsing the first two moves in our example.

We can see the actions assigning values to the attributes associated with nodes of the parse tree.

This leaf corresponds to the seq->begin production that initializes the coordinates, and these leaves correspond to the instruction productions that set dx and dy.

The interior nodes are calculated by applying the addition rules.

The root represents the final location of the robot after the parse.



Tree Traversal

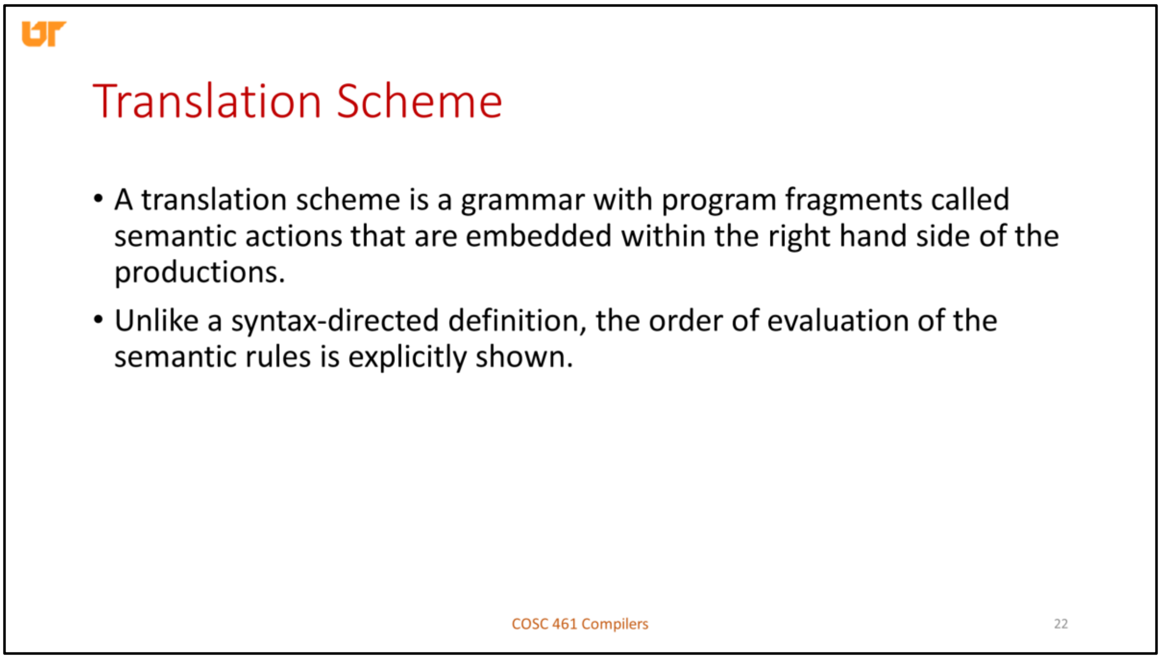
```
procedure visit(node N) {
    for ( each child C of N, from left to right ) {
        visit(C);
    }
    evaluate semantic rules at node N;
}
```

Fig. 2.11: A depth-first traversal of a tree

A syntax-directed definition does not impose any specific order for the evaluation of attributes on a parse tree. So, we use a tree traversal algorithm to evaluate the attributes at each node. A tree traversal of a tree starts at the root and visits each node of the tree in some order.

If all attributes are synthesized, then we can do a depth-first tree traversal to find the values of all attributes.

This algorithm shows a depth-first traversal since it visits the children of a node before evaluating the rules at the current node.



Translation Scheme

- A translation scheme is a grammar with program fragments called semantic actions that are embedded within the right hand side of the productions.
- Unlike a syntax-directed definition, the order of evaluation of the semantic rules is explicitly shown.

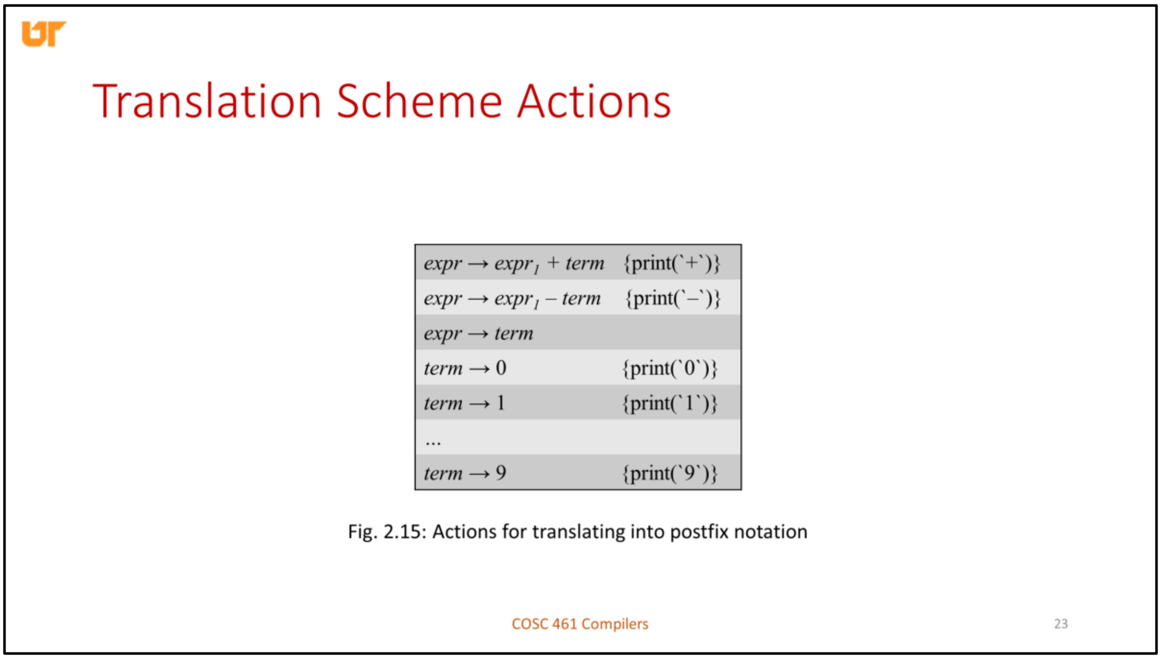
Now, I want to introduce a new way of doing syntax-directed translation. The way we just saw builds translations by attaching strings as attributes to nodes in the parse tree.

Another way to build up a translation is to use a translation scheme. A translation scheme attaches program fragments called semantic actions to the RHS of the productions of a grammar. These program fragments are executed when that production is used during parsing.

The combined result of all these fragment executions, in the order induced by the parsing process, produces the translation of the program under consideration.

So, a translation scheme is very similar to a syntax-directed definition. The main difference is that, with a translation scheme, you specify the order of evaluation of the semantic rules.

Let's look at an example.



Translation Scheme Actions

$expr \rightarrow expr_1 + term$	{print(`+`)}
$expr \rightarrow expr_1 - term$	{print(`–`)}
$expr \rightarrow term$	
$term \rightarrow 0$	{print(`0`)}
$term \rightarrow 1$	{print(`1`)}
…	
$term \rightarrow 9$	{print(`9`)}

Fig. 2.15: Actions for translating into postfix notation

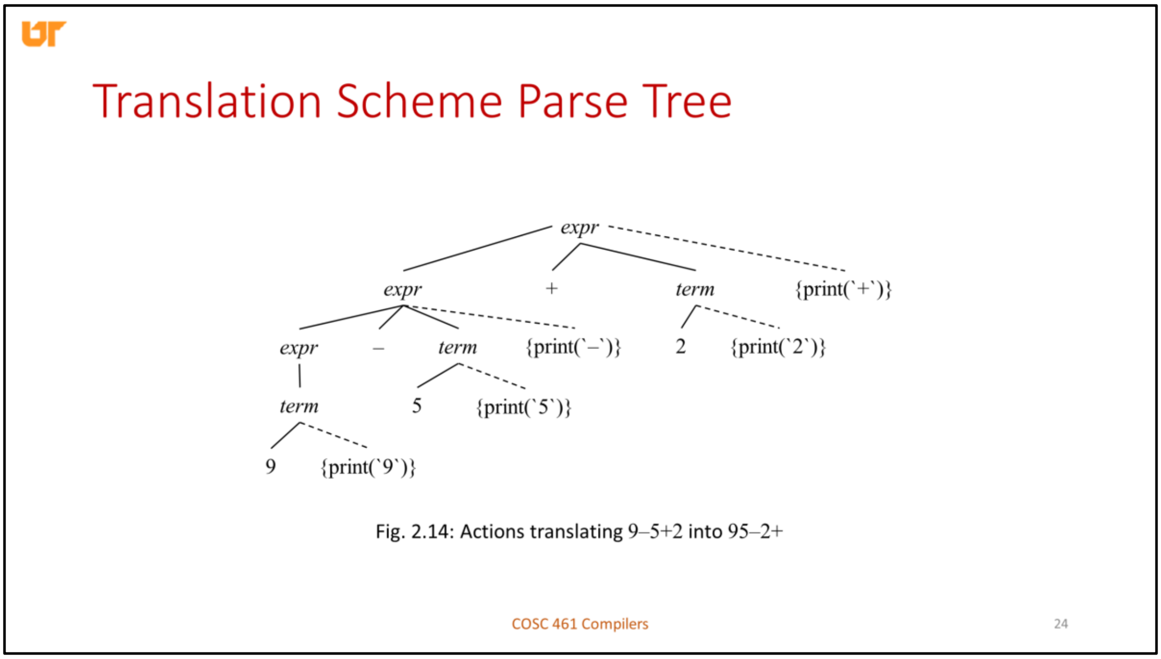
COSC 461 Compilers 23

Here is the same grammar with a translation scheme for converting infix to postfix attached.

The actions taken to do the translation are shown between curly braces.

So – in this example – we just use print statements to print out our translation.

Now, in this example, all of the print statements come at the end of the production – but they do not need to. You could put the print statement anywhere in this production. For instance, 'print +' could go before $expr_1$ in the first production if you'd like – but that would have implications for how the translation is done.



Translation Scheme Parse Tree

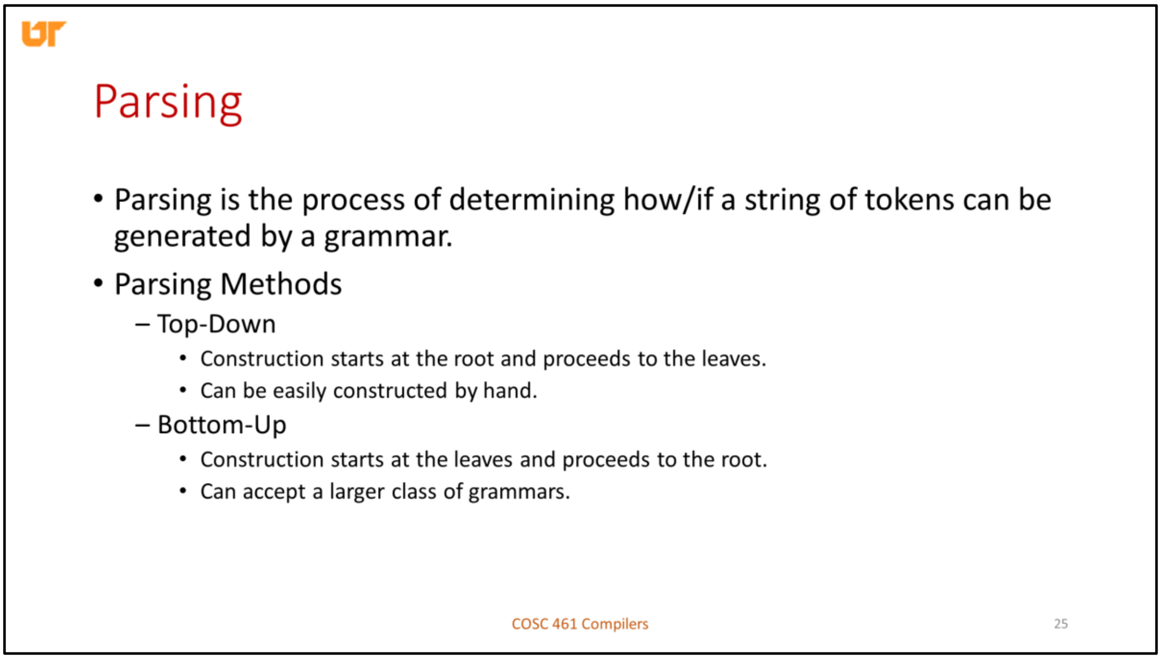
Fig. 2.14: Actions translating 9–5+2 into 95–2+

COSC 461 Compilers 24

So, when we do a parse of the string ‘9-5+2’ with this grammar – we get this parse tree.

And, when we do a depth-first traversal of the tree, we print the postfix output ‘95-2+’. You can see this just by reading the print actions from left to right, which happens to be the depth-first order.

How could we modify the translation scheme in Figure 2.15 to cause infix to be printed out?
We could move the print actions after the + and -, and still use a depth-first traversal to evaluate the translation.



Parsing

- Parsing is the process of determining how/if a string of tokens can be generated by a grammar.
- Parsing Methods
  - Top-Down
    - Construction starts at the root and proceeds to the leaves.
    - Can be easily constructed by hand.
  - Bottom-Up
    - Construction starts at the leaves and proceeds to the root.
    - Can accept a larger class of grammars.

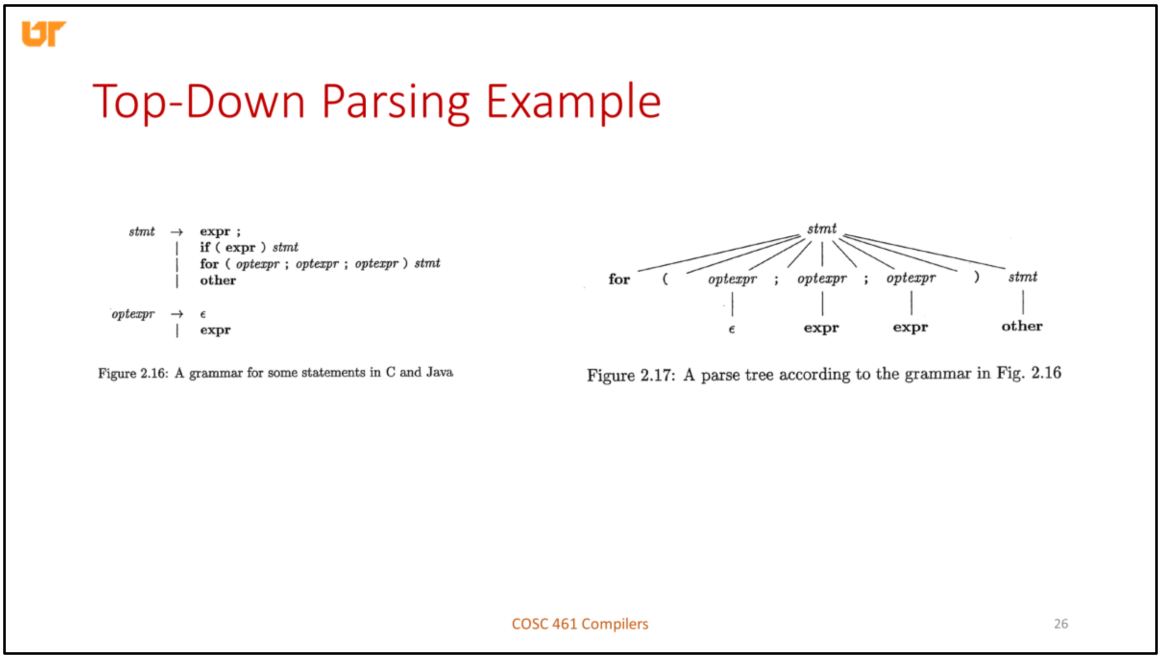
So far, we have mostly discussed what parsing is and how we can use it to do a translation. Now, I want to talk a little bit about how we actually do parsing. We'll discuss this in much more detail in Chapter 4.

Remember that parsing is the process of determining how or if a string of tokens can be generated by a grammar.

Parsing methods fall into one of two general classes:

Top-down parsers – where the construction starts at the root and then proceeds to the leaves. This kind of parsing can be constructed easily by hand.

And we also have bottom-up parsers – where the construction starts at the leaves and proceeds to the root. These parsers are harder to construct by hand, but can also accept a larger class of grammars. So, these are used more often in tools that generate parsers from grammars.



Top-Down Parsing Example

```
stmt    →  expr ;
        |  if ( expr ) stmt
        |  for ( optexpr ; optexpr ; optexpr ) stmt
        |  other

optexpr →  ε
        |  expr
```

Figure 2.16: A grammar for some statements in C and Java

Figure 2.17: A parse tree according to the grammar in Fig. 2.16

To understand how top-down parsing works, I'd like to go through a simple example.

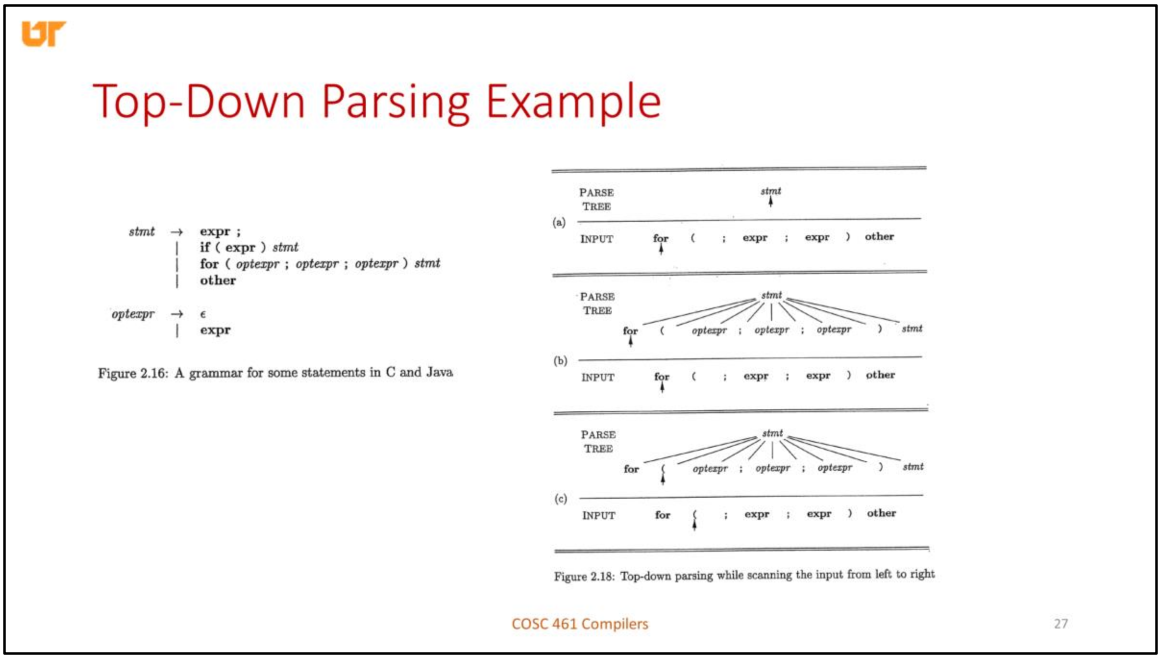
This grammar generates a subset of the statements in a language like C or Java. Figure 2.17 shows the corresponding parse tree for this grammar when given a for statement as input.

Constructing this tree using top-down parsing is straightforward.

You keep a lookahead symbol that points to the current token in your input stream. Then, starting from the start symbol, you expand the start symbol with a production that matches the current lookahead symbol. Then, you examine the child nodes you just added to the parse tree and advance the lookahead from left to right.

At terminal symbols, you just check to make sure they match the lookahead. If they do, just advance the lookahead and move to the next node. And at non-terminal symbols, you repeat the process by selecting a production to expand the non-terminal symbol.

The main point is that the tree is being constructed from the root to the leaves in a top-down fashion.



Top-Down Parsing Example

```
stmt    →  expr ;
        |  if ( expr ) stmt
        |  for ( optexpr ; optexpr ; optexpr ) stmt
        |  other

optexpr →  ε
        |  expr
```

Figure 2.16: A grammar for some statements in C and Java

(a)
PARSE TREE
stmt
INPUT
for (; expr ; expr) other
(b)
PARSE TREE
stmt
for (optexpr ; optexpr ; optexpr) stmt
INPUT
for (; expr ; expr) other
(c)
PARSE TREE
stmt
for (optexpr ; optexpr ; optexpr) stmt
INPUT
for (; expr ; expr) other

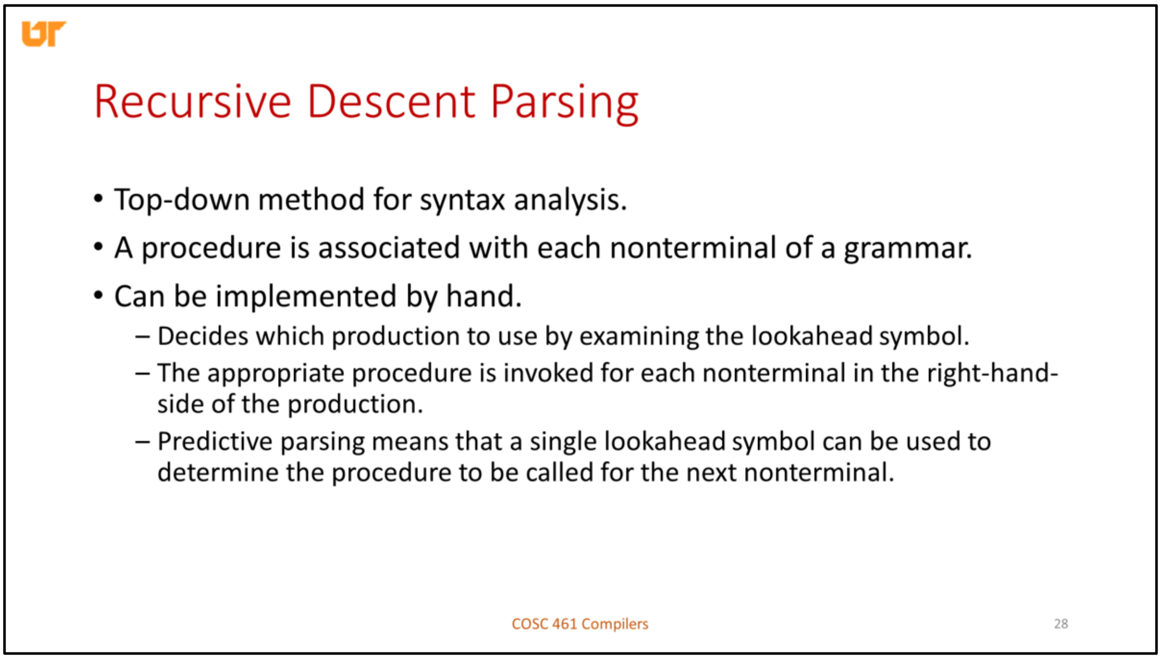
Figure 2.18: Top-down parsing while scanning the input from left to right

This figure just shows the first few steps of the parse that I just described.

In (a), we are considering how to expand the start symbol ‘stmt’, and the lookahead points to ‘for’. So, we select the production that starts with ‘for’, and expand the parse tree (b). Then, we start matching the children we added to the parse tree with the lookahead symbol – scanning left to right (c).

If some productions start with the same symbol, we might select the wrong production and have to backtrack. In this example, the grammar allows for what is called ‘predictive parsing’, and no backtracking is required.

Basically, predictive parsing means that the lookahead symbol will always unambiguously determine the next production to select when expanding a non-terminal. This grammar is suitable for predictive parsing, but not every grammar will be.



Recursive Descent Parsing

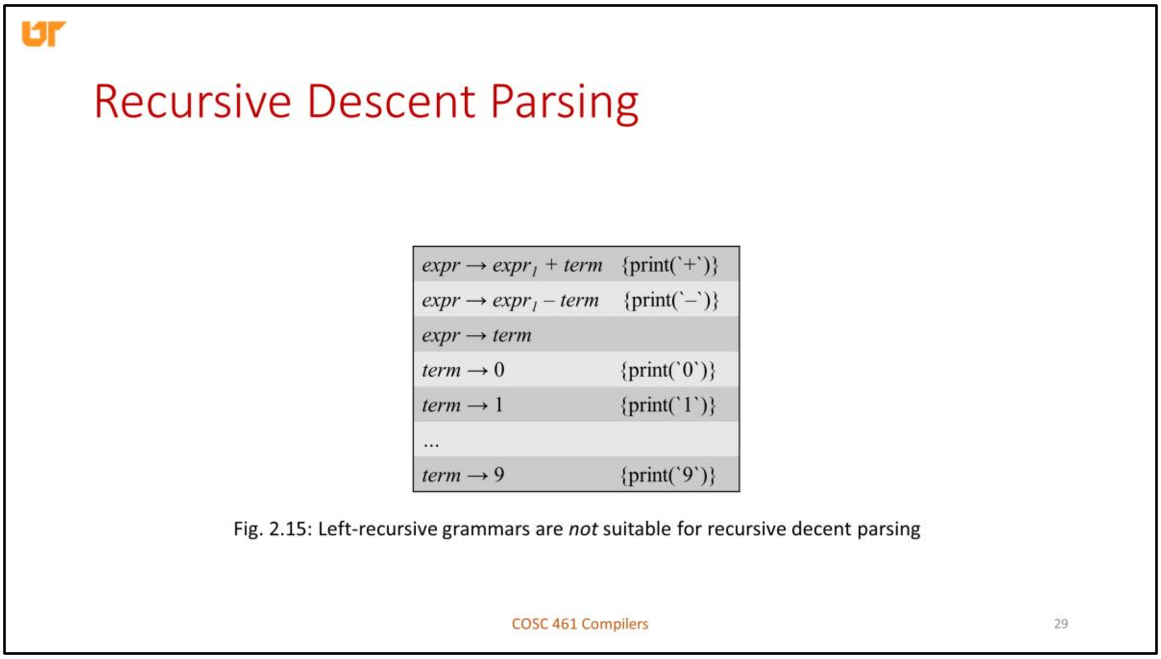
- Top-down method for syntax analysis.
- A procedure is associated with each nonterminal of a grammar.
- Can be implemented by hand.
  - Decides which production to use by examining the lookahead symbol.
  - The appropriate procedure is invoked for each nonterminal in the right-hand-side of the production.
  - Predictive parsing means that a single lookahead symbol can be used to determine the procedure to be called for the next nonterminal.

What I've described is actually a simple form of what is called recursive descent parsing. This style of parsing is very popular since it can easily be accomplished by hand.

Writing a recursive descent parser for a grammar is actually pretty easy.

First, you just create a procedure for each non-terminal of a grammar. You point your lookahead symbol to the start of the input string. Then, you decide which production to use by examining the lookahead symbol. Then, you just invoke the appropriate procedure for each non-terminal in the RHS of the production.

Also, I mentioned predictive parsing before. This just means that a single lookahead symbol can be used to determine the procedure to be called for the next non-terminal.



Recursive Descent Parsing

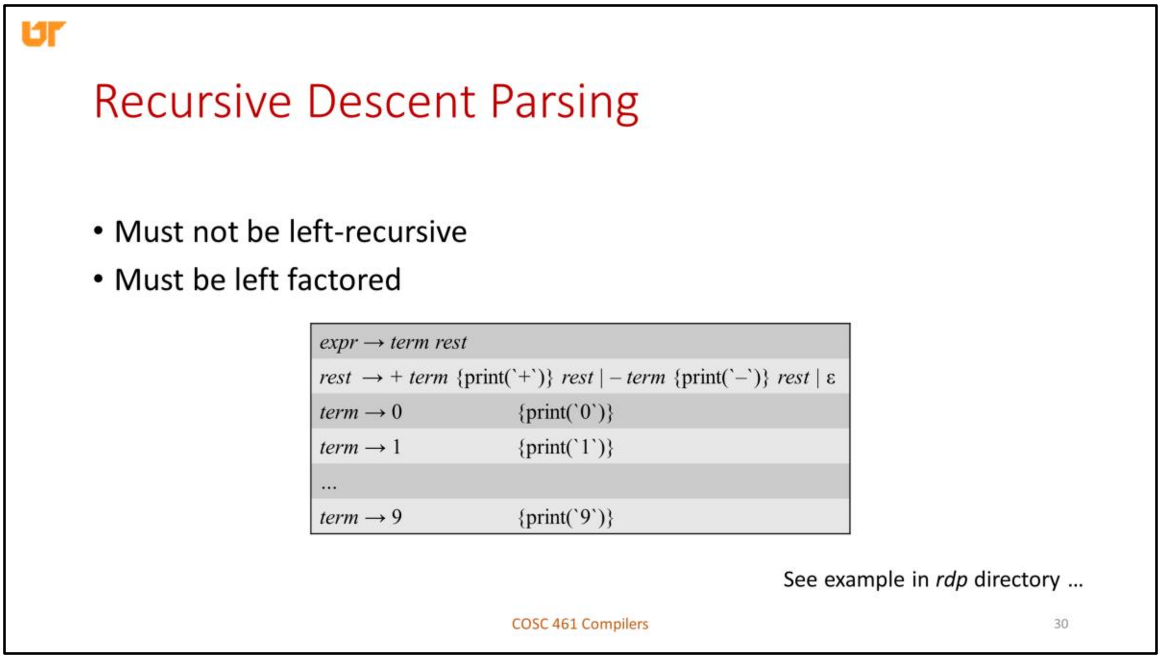
$expr \rightarrow expr_1 + term$	{print(`+`)}
$expr \rightarrow expr_1 - term$	{print(`–`)}
$expr \rightarrow term$	
$term \rightarrow 0$	{print(`0`)}
$term \rightarrow 1$	{print(`1`)}
…	
$term \rightarrow 9$	{print(`9`)}

Fig. 2.15: Left-recursive grammars are *not* suitable for recursive decent parsing

One problem with recursive descent parsing is that it is not suitable for every grammar. In particular, if your grammar is left recursive, then you cannot use recursive descent parsing.

This is an example of a left-recursive grammar. A left-recursive grammar is simply one where the leftmost symbol of the body of a production is the same as the non-terminal at the head of the production

If we used recursive descent parsing on this grammar, then the parser would go into an infinite loop.



Recursive Descent Parsing

- Must not be left-recursive
- Must be left factored

expr → *term rest*

rest → + *term* {print(`+`)} *rest* | – *term* {print(`–`)} *rest* | ε

term → 0 {print(`0`)}

term → 1 {print(`1`)}

…

term → 9 {print(`9`)}

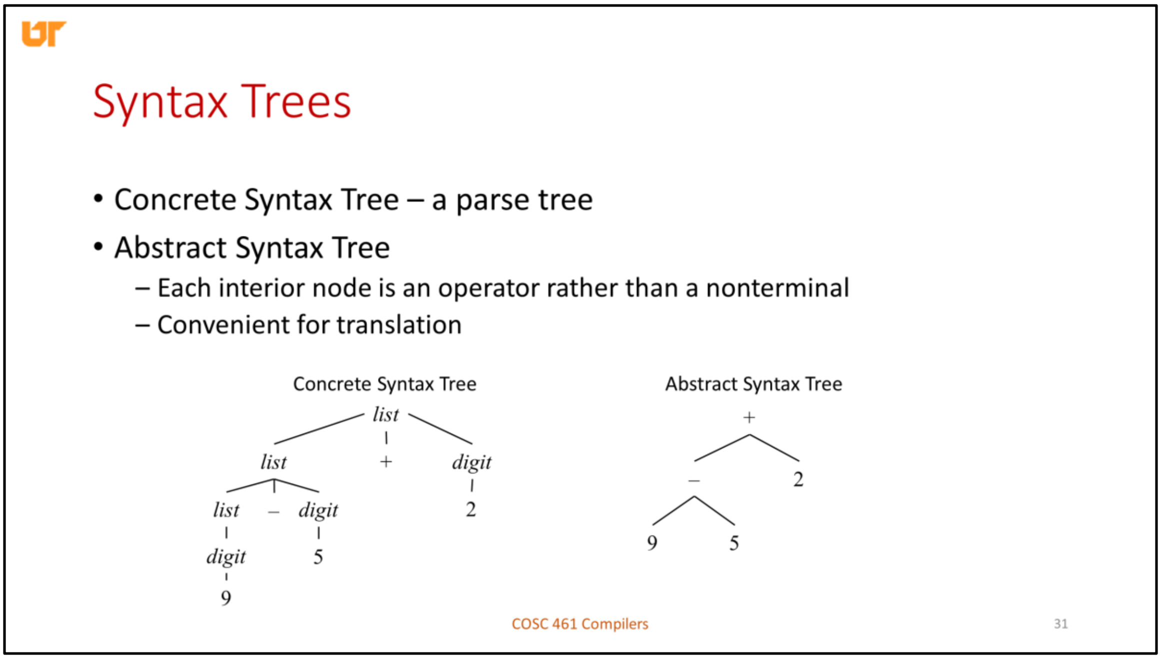
See example in *rdp* directory …

We can transform grammars that are left-recursive to be suitable for recursive descent parsing using a process called left factoring. You'll learn how to left factor a grammar later in the course.

This figure shows how we could left factor the figure from the previous slide to make it suitable for recursive descent parsing. Notice that there are no recursive non-terminals on the leftmost side of the bodies of the productions.

Write it up on the board.

OK – so I have an actual implementation of a recursive descent parser written in C that I can show you. I'll post some of these examples on the course website when I get a chance.



Syntax Trees

- Concrete Syntax Tree – a parse tree
- Abstract Syntax Tree
  - Each interior node is an operator rather than a nonterminal
  - Convenient for translation

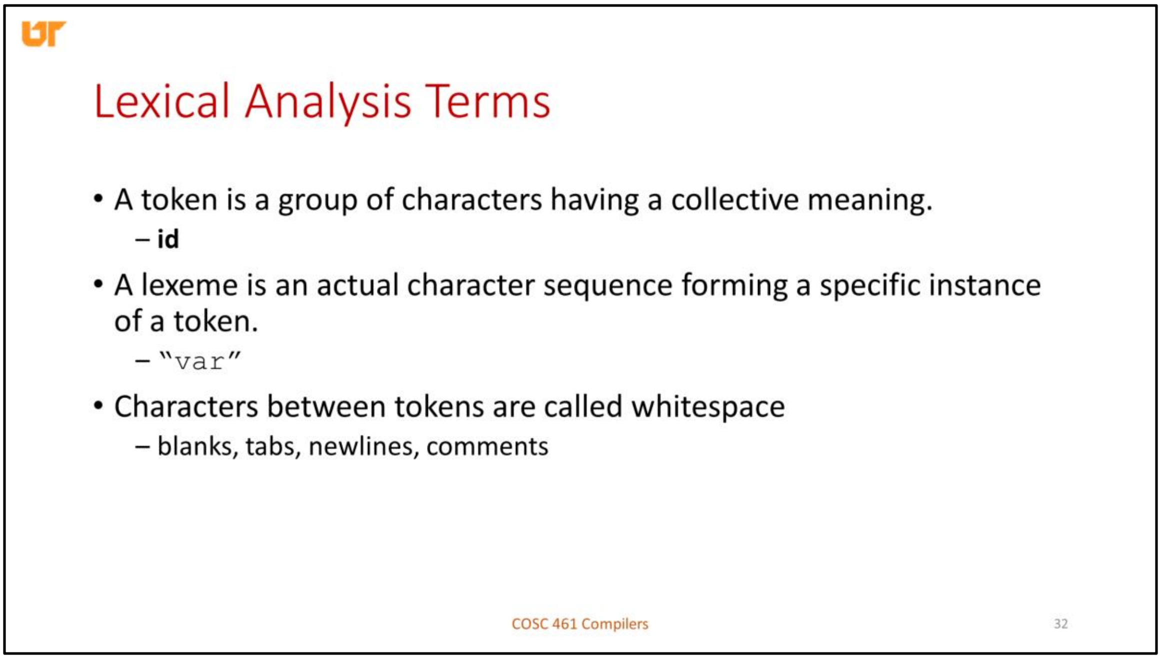
OK – before moving on, I want to quickly discuss another useful concept for translation called abstract syntax trees.

A concrete syntax tree is just a parse tree – we've already seen several examples of these.

An abstract syntax tree is like the parse trees that we’ve already seen with the difference that each interior node is an operator rather than a non-terminal.

This is useful for translation because the tree can be traversed and code can be emitted just like with a parse tree or ‘concrete syntax tree’.

However, parse trees often contain several ‘helper’ nodes for single productions, such as the production list -> digit in this example. This extra information is necessary to show the parsing process, but is not required for translation. So, this information is omitted in AST’s, which makes them simpler and easier to work with during translation.



Lexical Analysis Terms

- A token is a group of characters having a collective meaning.
  - **id**
- A lexeme is an actual character sequence forming a specific instance of a token.
  - `“var”`
- Characters between tokens are called whitespace
  - blanks, tabs, newlines, comments

COSC 461 Compilers 32

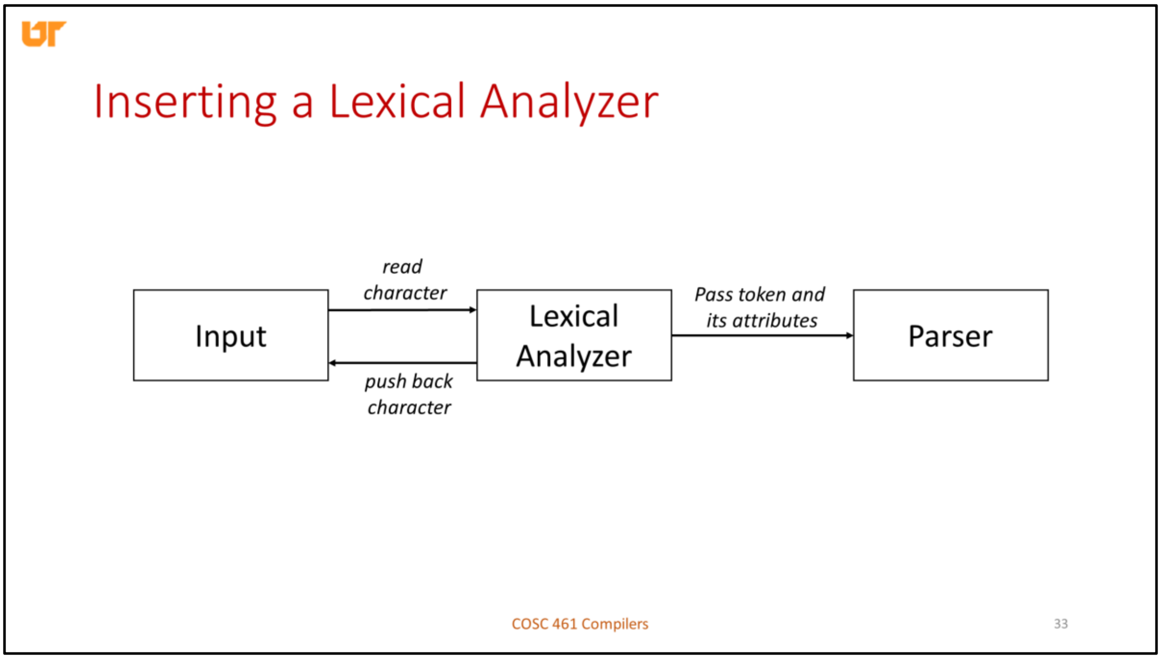
OK – now that I’ve discussed a little about how parsing works – let’s talk about how lexical analysis works. First, I’d like to go over a few terms.

A lexical analyzer reads characters from the input and groups them into tokens. A token is just a group of characters that have a collective meaning. For instance, this might be a terminal corresponding to identifiers in your source code. A lexeme is an actual character sequence forming a specific instance of a token. For instance, if you have the sequence “var” that refers to a variable in your source code, this sequence might form a lexeme of the token ‘id’.

Tokens often carry additional information in the form of attribute values. For example, all numerical values may be represented by the same token num, and the attribute value will denote the actual numeral in each case.
So, for instance, if you have the string '88' in your program. '88' is the lexeme. The token for that lexeme might be **num**, but the value attribute for that token is the literal number 88.

Characters between tokens are called whitespace. In most programming languages, whitespace serves as a delimiter between tokens.

We will discuss lexical analysis in more detail when we begin the next chapter.



Inserting a Lexical Analyzer

Input

read character

push back character

Lexical Analyzer

Pass token and its attributes

Parser

We typically insert a lexical analyzer between the input stream and the parser. As we will see later in the course, it is much easier to break the parsing process down into lexical and syntax analysis.

Also, similar to parsing, we will often find that the lexical analyzer needs one character of lookahead to decide on the token that should be returned.

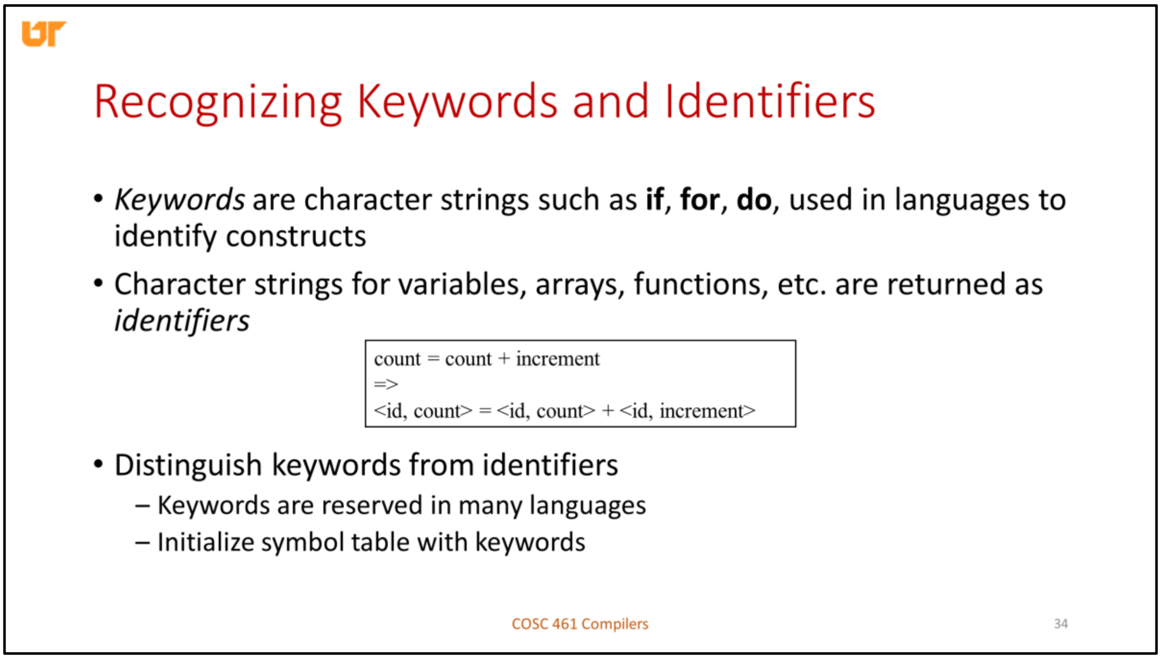
For example, after reading a > sign, we need a lookahead character to decide if the token is >= or >.

Thus, we also need the ability to be able to push back a single character onto the input stream if the input character we have read is not utilized.

For this reason and to make the process of reading the input more efficient, most lexical analyzers use input buffers, and fetch a block of characters at a time.

I/O is typically buffered. Even from standard input, we read a line at a time.

buffer.c example



Recognizing Keywords and Identifiers

- *Keywords* are character strings such as **if**, **for**, **do**, used in languages to identify constructs
- Character strings for variables, arrays, functions, etc. are returned as *identifiers*

```
count = count + increment
=>
<id, count> = <id, count> + <id, increment>
```

- Distinguish keywords from identifiers
  - Keywords are reserved in many languages
  - Initialize symbol table with keywords

Part of what the lexical analyzer does is recognize keywords and identifiers.

Keywords are character strings …

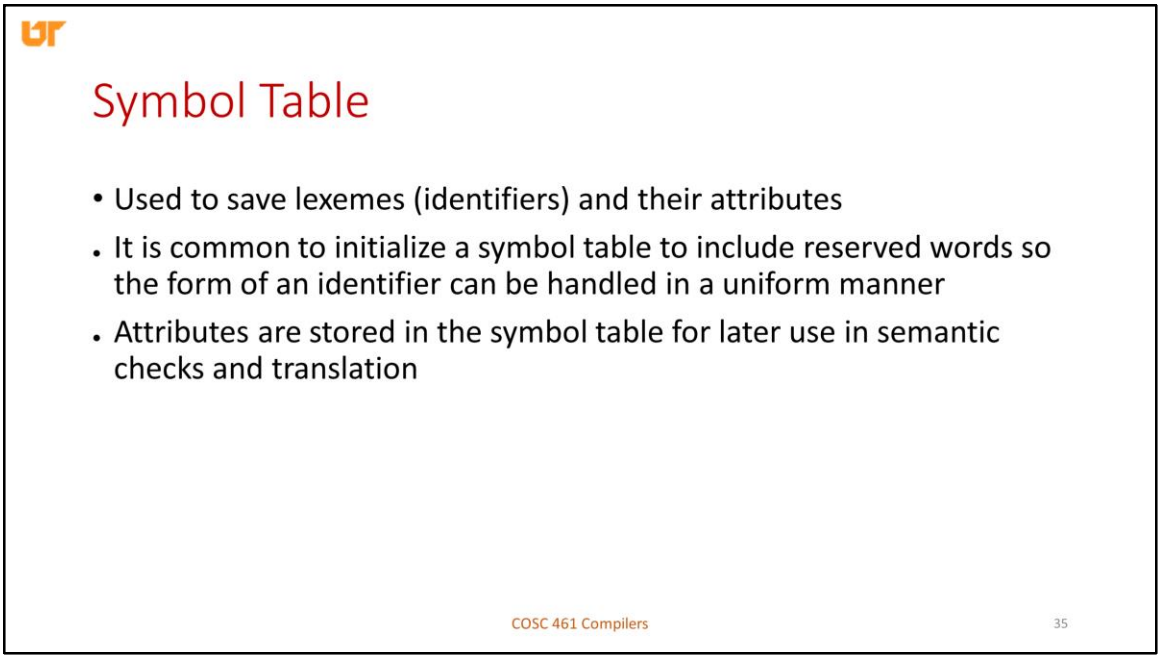
Character strings for variables, arrays, functions, and other program symbols have to be distinguished from keywords and returned as identifiers.

So, in this example, the lexical analyzer will create three tokens for the three identifiers in the statement. Each token has an attribute value that corresponds to the lexeme that comprises that instance of the identifier.

Keywords are reserved in many languages and cannot be used as identifiers. To reserve the key words, compilers will simply initialize the symbol table with the keywords to indicate those words are reserved.

I have an example of a very simple lexical analyzer written in C.

Show lexan.c



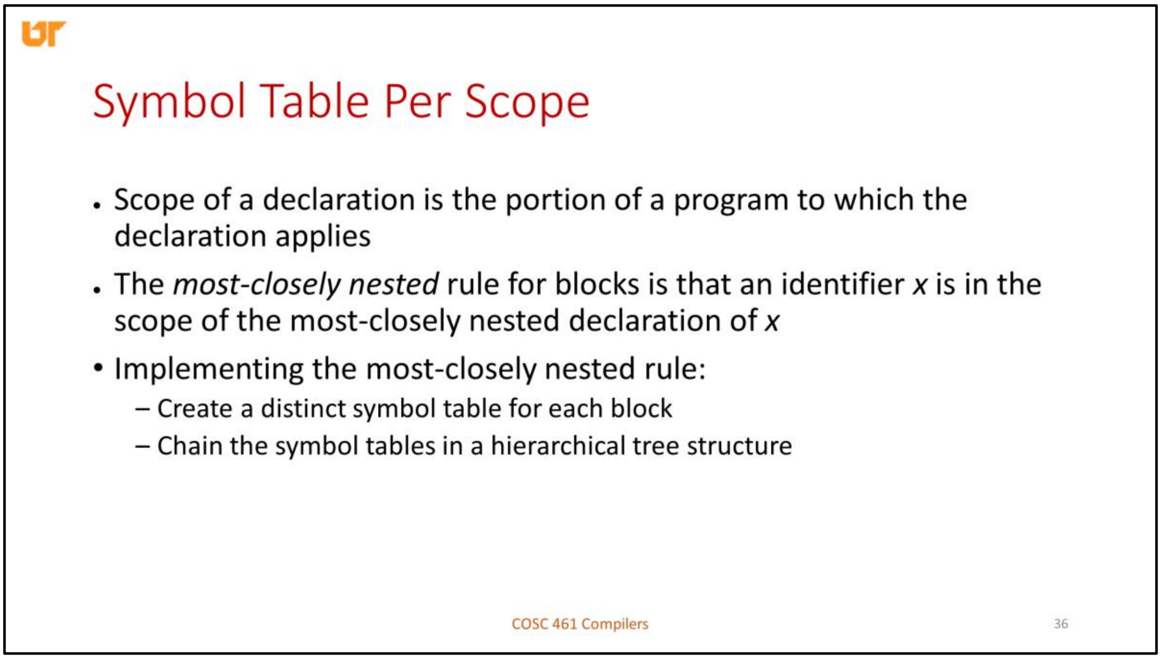
Symbol Table

- Used to save lexemes (identifiers) and their attributes
- It is common to initialize a symbol table to include reserved words so the form of an identifier can be handled in a uniform manner
- Attributes are stored in the symbol table for later use in semantic checks and translation

A symbol table is a data structure containing a record for each identifier and fields for their attributes.
Symbol tables are often initialized to include several reserved words so reserved words and identifiers can be handled uniformly.

The compiler will update the symbol table during lexical analysis and parsing. Then, it will continue to use the information in the table during semantic analysis and intermediate code generation.

Symbol tables also store attributes for each identifier to help during these later phases. For instance, the symbol table will store type information for each identifier so that the semantic analysis can determine if or when type coercion is necessary.



Symbol Table Per Scope

- Scope of a declaration is the portion of a program to which the declaration applies
- The *most-closely nested* rule for blocks is that an identifier *x* is in the scope of the most-closely nested declaration of *x*
- Implementing the most-closely nested rule:
  - Create a distinct symbol table for each block
  - Chain the symbol tables in a hierarchical tree structure

For most languages, symbol tables require a notion of scope. The scope of a declaration is the portion of the program to which the declaration applies.

Q. So, what's an example of scope in C?
A. We can talk about program, file, function, block, and prototype scope.

Q. What is the scope of globals?
A. Program scope

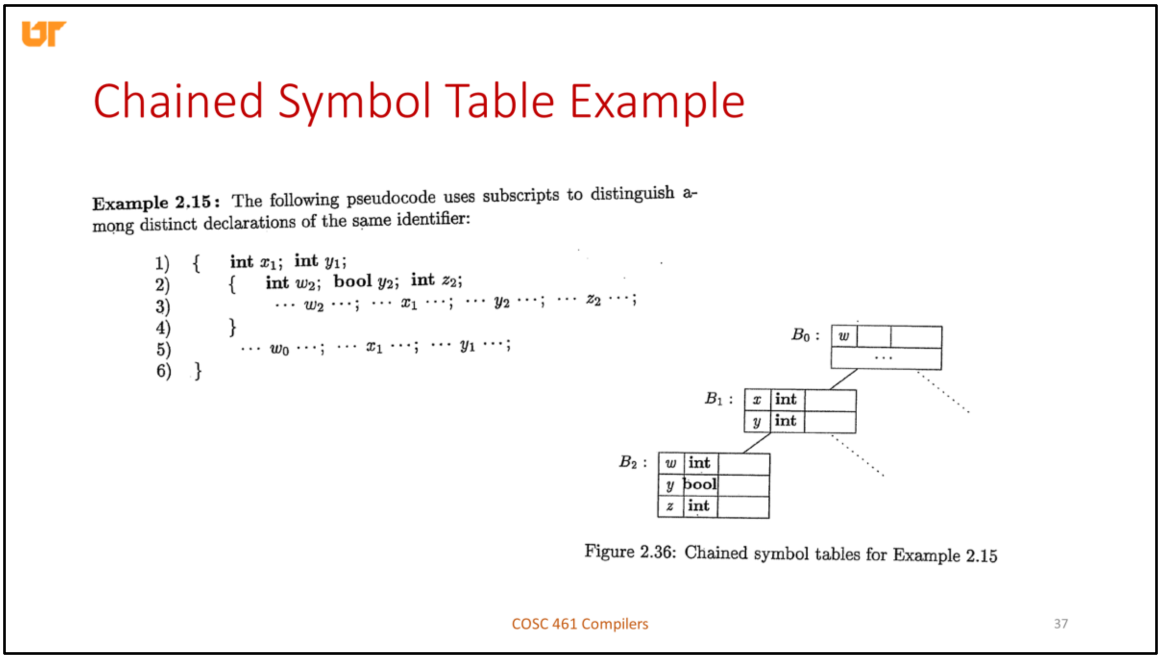
Scope is distinct from visibility. Visibility refers to whether or not a declaration is visible or accessible at some point in the program. Here's an example where scope might be different than visibility.

```
int x = 0; // has file scope
void *foo(int bar) {
  int x = bar; // has function scope, outer x is not visible here
  return (x << 2);
}
```

To deal with these situations, we have the most-closely nested rule. This says an identifier x is in the scope of the most-closely nested declaration of x.

To implement the most closely nested rule, we create a distinct symbol table for each block in the program and chain the symbol tables using a hierarchical tree structure.

Note that there is no need to create a new symbol table for a block where there are no variable declarations.
To search for a variable in the chained symbol table, the search starts from the current symbol table and proceeds along the chain of ancestors of the table.



Chained Symbol Table Example

Example 2.15: The following pseudocode uses subscripts to distinguish among distinct declarations of the same identifier:

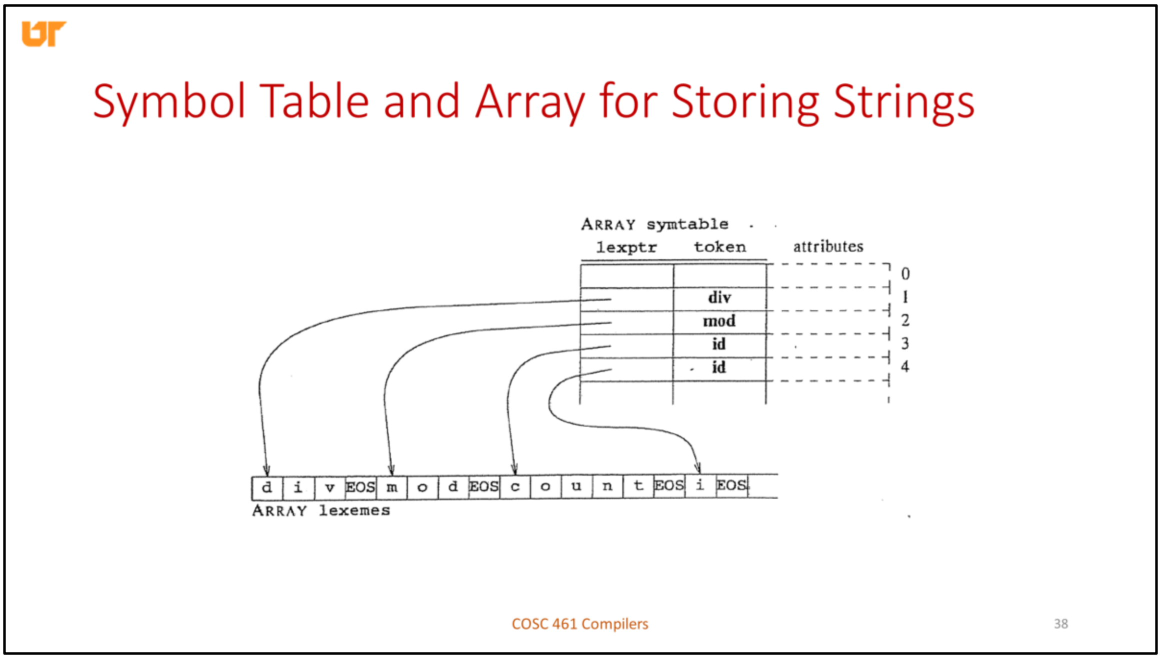
```
1) {   int $x_1$; int $y_1$;
2)       {   int $w_2$; bool $y_2$; int $z_2$;
3)             ⋯ $w_2$ ⋯; ⋯ $x_1$ ⋯; ⋯ $y_2$ ⋯; ⋯ $z_2$ ⋯;
4)       }
5)         ⋯ $w_0$ ⋯; ⋯ $x_1$ ⋯; ⋯ $y_1$ ⋯;
6) }
```

Figure 2.36: Chained symbol tables for Example 2.15

This example shows how a chained symbol table works.

In this example, we have built a chained symbol table, with B1 corresponding to the block starting on line 1, and B2 corresponding to the block starting on line 2.

When control is between lines 2 to 4, the environment is represented by a reference to the symbol table B2. So, in lines 2-4, variable x in B1 is visible, but w from block B0, and y from block B1 are hidden due to the identical identifiers declared in block B2.



Symbol Table and Array for Storing Strings

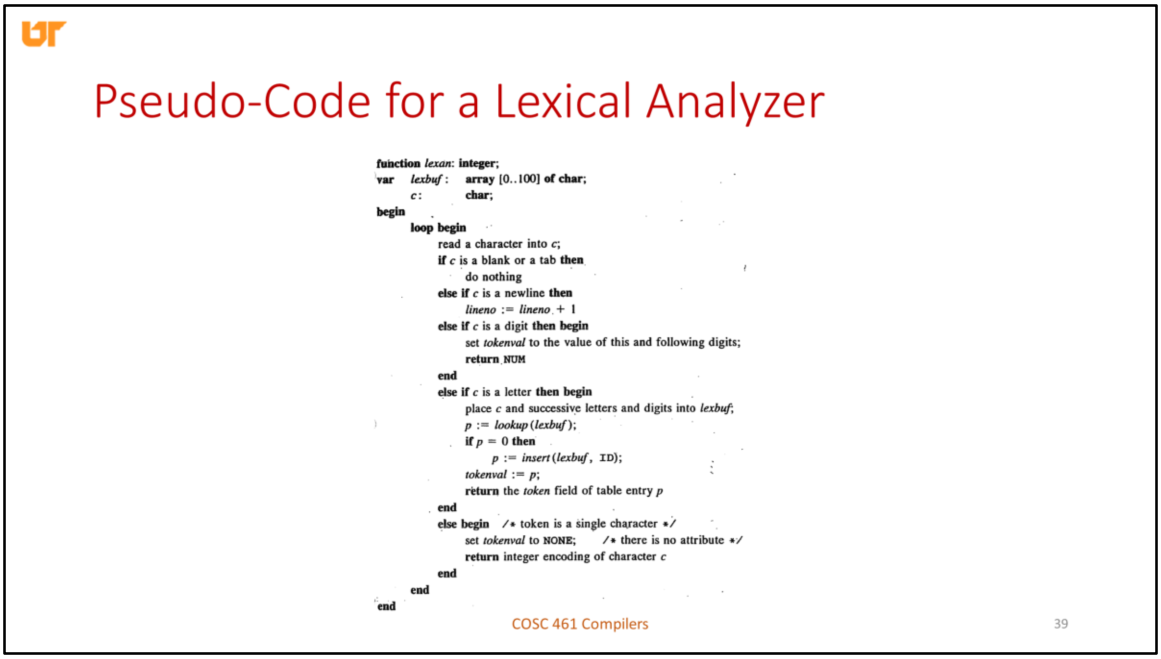
We often represent as a symbol table as an array of pointers, tokens, and possibly the token's attributes. The pointers in the symbol table point to the lexeme strings in another array. We organize the table this way in order to save space and make it easier to traverse the symbol table.

This approach avoids having space for a large string for each symbol in the symbol table, but it does require an extra dereference to find the location in the string table.

Usually, the symbol table is not a fixed size table.

Programs can have as many identifiers as needed (though there has to be some limit due to memory constraints). So compilers usually use some other form such as a linked list with hashing for fast access.

We will see an example of this later in the course.



Pseudo-Code for a Lexical Analyzer

```
function lexan: integer;
var  lexbuf :  array [0..100] of char;
     c :       char;
begin
    loop begin
        read a character into c;
        if c is a blank or a tab then
            do nothing
        else if c is a newline then
            lineno := lineno + 1
        else if c is a digit then begin
            set tokenval to the value of this and following digits;
            return NUM
        end
        else if c is a letter then begin
            place c and successive letters and digits into lexbuf;
            p := lookup(lexbuf);
            if p = 0 then
                p := insert(lexbuf, ID);
            tokenval := p;
            return the token field of table entry p
        end
        else begin  /* token is a single character */
            set tokenval to NONE;     /* there is no attribute */
            return integer encoding of character c
        end
    end
end
```

We earlier looked at source code for a simple lexical analyzer for parsing numbers.

This figure shows pseudo-code for a simple lexical analyzer that also parses identifiers.

The function returns a single token. So, it's expected you would call this in a loop during program parsing.

So, we can see it just skips over tabs and newlines.
We use lineno to mark what line we're on for reporting error messages and debugger information.

So, if we match a digit, then we return a number.

Else, if we match a character, then we're at either an identifier or a reserved word, and we put that character and the following letters and digits into lexbuf.

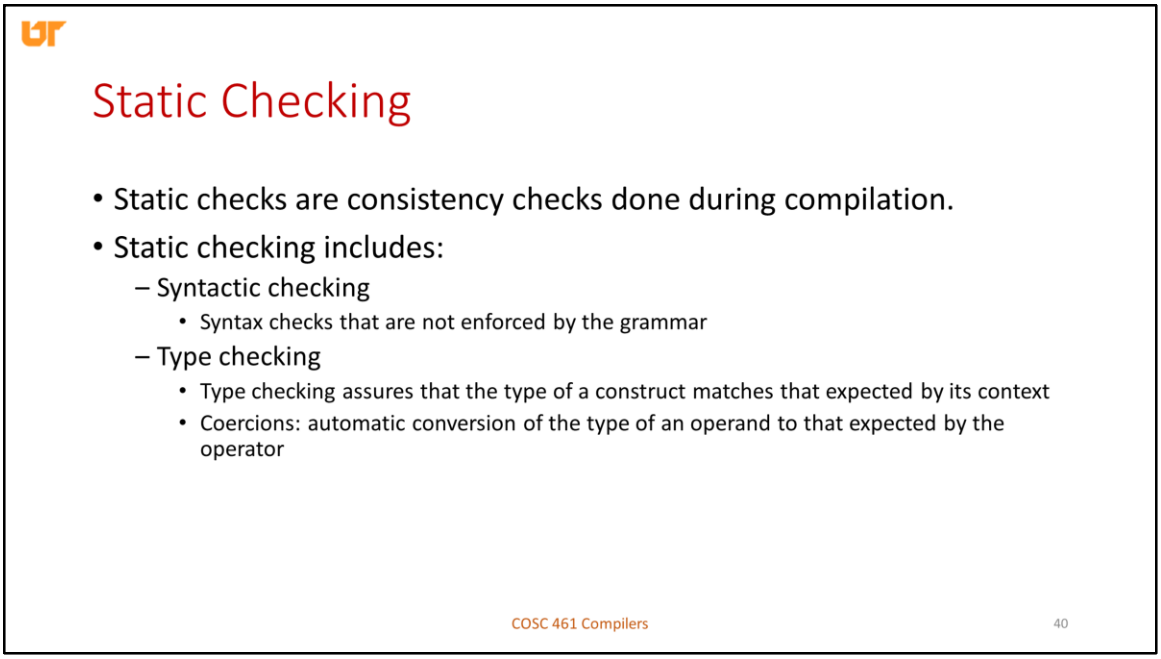
Then, we check if the lexeme is already in the symbol table. The lookup() function is just checking if the lexeme is in the symbol table.

If it's not, then we put the identifier into the symbol table. After that line, p is the entry in the symbol table corresponding to the token we just matched, and we return the token field of the symbol table entry p.

Note, that the token field of the word might be a reserved word, so we can't just always return an ID.

Also, note that, even if lookup returns non-zero, then the lexical analyzer does not produce an error.

This is because it is the parser's job to find errors if (a) variable is used before declaration, or (b) declared twice in the same scope.



Static Checking

- Static checks are consistency checks done during compilation.
- Static checking includes:
  - Syntactic checking
    - Syntax checks that are not enforced by the grammar
  - Type checking
    - Type checking assures that the type of a construct matches that expected by its context
    - Coercions: automatic conversion of the type of an operand to that expected by the operator

Now, I want to talk briefly about static checking. Static checks are just consistency checks that are done during compilation.

They assure that a program can be compiled successfully, but they also do things other than simple lexical analysis or parsing. Compilers use static checking to try to catch programming errors early on.

For instance, with syntactic checking, the compiler will perform some syntax checks that aren't necessarily enforced by the grammar. This can include things like checking if a variable has been declared at most once in a certain scope, or that a break statement has an enclosing loop or switch statement.

Another type of static checking is type checking. Type checking assures that the type of a construct matches what's expected by it's context.

The source language will often have type rules that assure that an operator or function is applied to the right number and type of operands.
For example, in this statement:

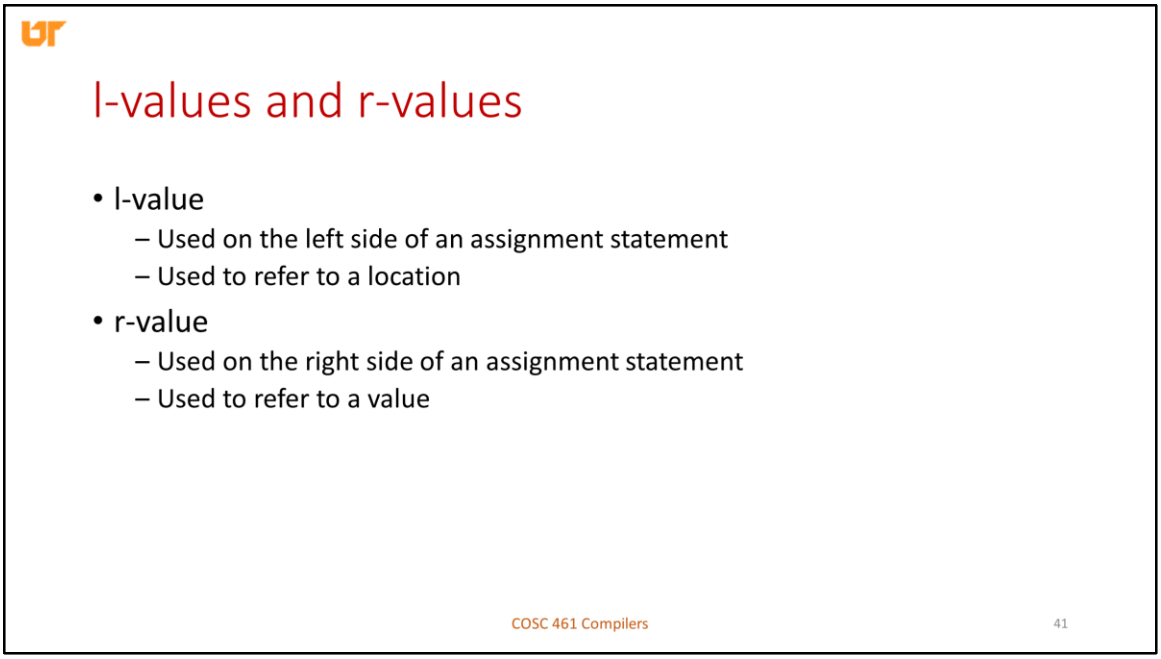
if(expr) statement, expr should have type Boolean

The type checker will check if expr has type Boolean.

And I mentioned coercion in a previous class. Coercion is just the automatic conversion of the type of an operand to be the type that is expected by the operator.

For example, in 2 * 5.45. 2 is an integer, 5.45 is a float. The compiler will automatically convert 2 to 2.0 so it can do the multiplication.

Why can we not mix integers and floats in this instruction?



l-values and r-values

- l-value
  - Used on the left side of an assignment statement
  - Used to refer to a location
- r-value
  - Used on the right side of an assignment statement
  - Used to refer to a value

One concept that is useful in static checking is l-values and r-values.

These things get their names by which they usually appear on in an assignment statement.

An l-value, which is usually on the left side of an assignment, is used to refer to a location.

And an r-value, which is usually on the right side of an assignment, is used to refer to a value.

If you think about it as with function calls, an l-value would be passed by reference and an r-value would be passed by value.

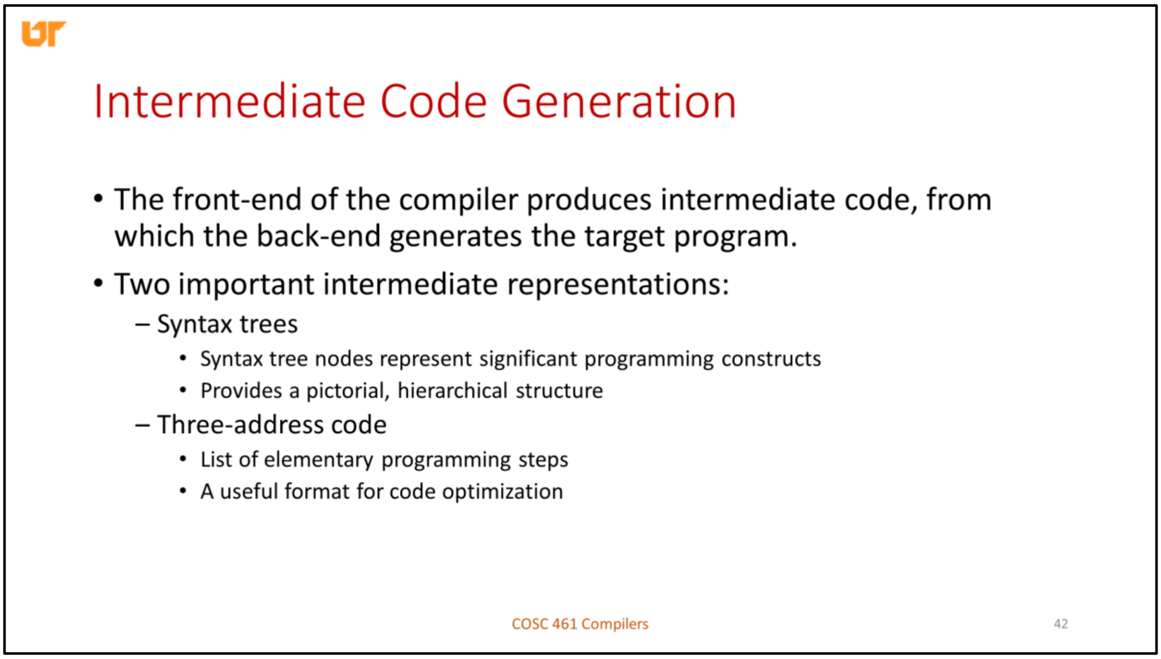
In this example:
i = i+1

i on the left side of the assignment refers to the location where you will store (i+1), while the i on the right side refers to the value in the location i.

The static checker will always try to assure that the left side of an assignment is always an l-value. However, l-values can appear on the right side of the assignment as well:

In this statement:
i = *p;

What is p?
p refers a pointer (location) or l-value that is dereferenced by the * operator.



Intermediate Code Generation

- The front-end of the compiler produces intermediate code, from which the back-end generates the target program.
- Two important intermediate representations:
  - Syntax trees
    - Syntax tree nodes represent significant programming constructs
    - Provides a pictorial, hierarchical structure
  - Three-address code
    - List of elementary programming steps
    - A useful format for code optimization

Now, I want to talk briefly about intermediate code generation before concluding chapter 2.

As we've seen, the front-end of the compiler produces intermediate code, from which the back-end generates the target program.

Q. How does intermediate code reduce the effort of producing a compiler?
A. Producing intermediate code reduces the effort required to retarget a compiler to a new machine.

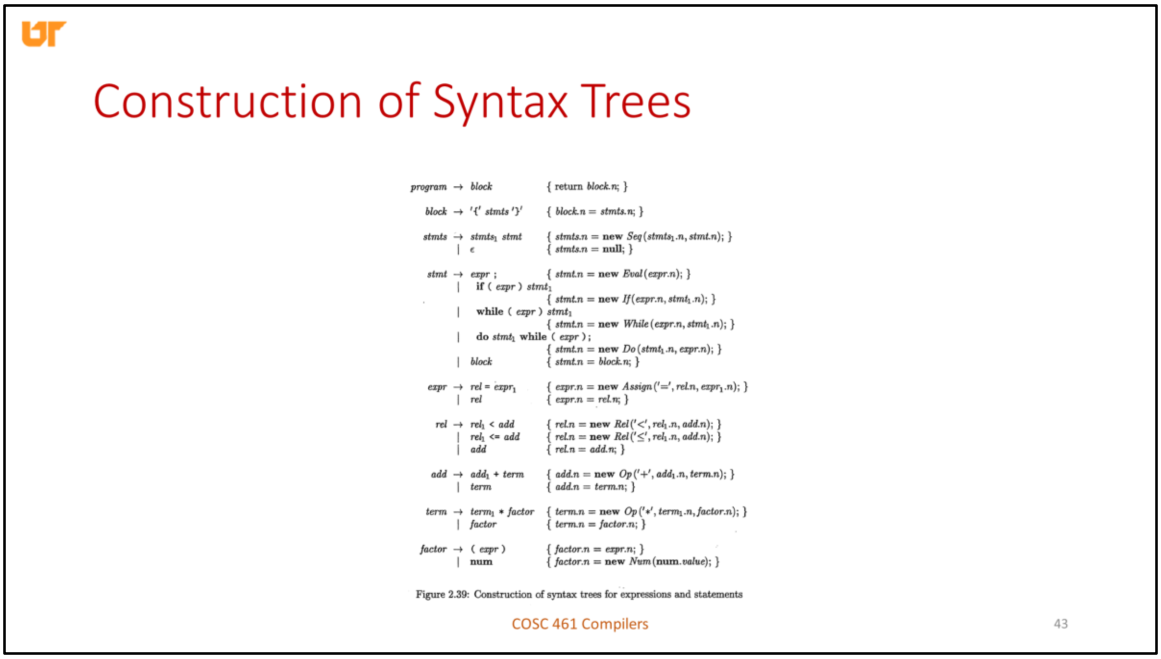
There are two important IR's that we'll look at:
The first is syntax trees. Syntax trees are very similar to all the parse trees we've seen up to this point. Actually, we saw an example of a syntax tree earlier (back on slide 31). These are just trees with nodes that represent significant programming constructs. They provide a pictorial, hierarchical structure of your program and how it will be evaluated.

Many compilers do not actually produce a syntax tree. Often they will produce another intermediate form, like three address code, and only implicitly produce the syntax-tree representation.

Three address code is simply a list of elementary programming steps. It's a useful format for doing code optimizations on the intermediate form. We'll see some examples of three address code in just a bit.

Another popular intermediate code representation is producing code for a stack machine. In a stack-based machine, operations are done by pushing operands onto a stack and then performing your operations on the operands on top of the stack – similar to evaluating postfix.
Java bytecodes is an example of this format.

We'll cover these concepts in more detail in chapter 6 of this course.



Construction of Syntax Trees

```
program → block             { return block.n; }

  block → '{' stmts '}'     { block.n = stmts.n; }

  stmts → stmts1 stmt       { stmts.n = new Seq(stmts1.n, stmt.n); }
        | ε                 { stmts.n = null; }

   stmt → expr ;            { stmt.n = new Eval(expr.n); }
        |  if ( expr ) stmt1
                            { stmt.n = new If(expr.n, stmt1.n); }
        |  while ( expr ) stmt1
                            { stmt.n = new While(expr.n, stmt1.n); }
        |  do stmt1 while ( expr );
                            { stmt.n = new Do(stmt1.n, expr.n); }
        | block             { stmt.n = block.n; }

   expr → rel = expr1       { expr.n = new Assign('=', rel.n, expr1.n); }
        | rel               { expr.n = rel.n; }

    rel → rel1 < add        { rel.n = new Rel('<', rel1.n, add.n); }
        | rel1 <= add       { rel.n = new Rel('≤', rel1.n, add.n); }
        | add               { rel.n = add.n; }

    add → add1 + term       { add.n = new Op('+', add1.n, term.n); }
        | term              { add.n = term.n; }

   term → term1 * factor    { term.n = new Op('*', term1.n, factor.n); }
        | factor            { term.n = factor.n; }

 factor → ( expr )          { factor.n = expr.n; }
        | num               { factor.n = new Num(num.value); }
```

Figure 2.39: Construction of syntax trees for expressions and statements

This figure shows a translation scheme for constructing a syntax tree during translation.

All the non-terminals in the translation scheme have an attribute *n*, which is a node of the syntax tree.

Many of the productions here will create a new node in the parse tree.

Just as we have syntax trees for expressions, where we have the operator as the parent, and the operands as the children, we can have syntax trees for program statements.

So, for example, we have the operator *. We use * like this:

term -> term1 * factor

It creates this syntax tree:

*

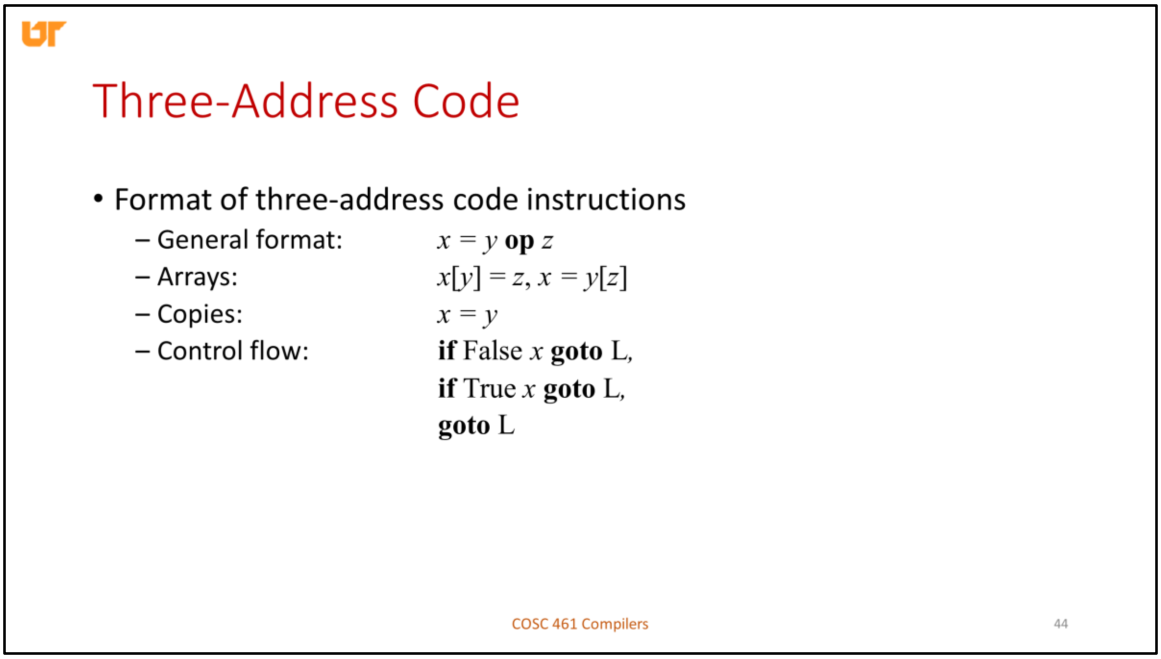
```
      /        \
   term1    factor
```

We can also have syntax trees for a statement like 'while'. the while statement looks like this:

while (expr) stmt

This translation scheme creates a tree that looks like this:

```
     while
   /      \
expr.n    stmt.n
```



Three-Address Code

- Format of three-address code instructions
  - General format: $x = y$ **op** z
  - Arrays: $x[y] = z$, $x = y[z]$
  - Copies: $x = y$
  - Control flow: **if** False x **goto** L,
if True x **goto** L,
goto L

COSC 461 Compilers 44

So, that's syntax trees. Now, let's look at three-address code.

Three address code consists of a sequence of instructions executed in numerical sequence.

The general format is x = y op z

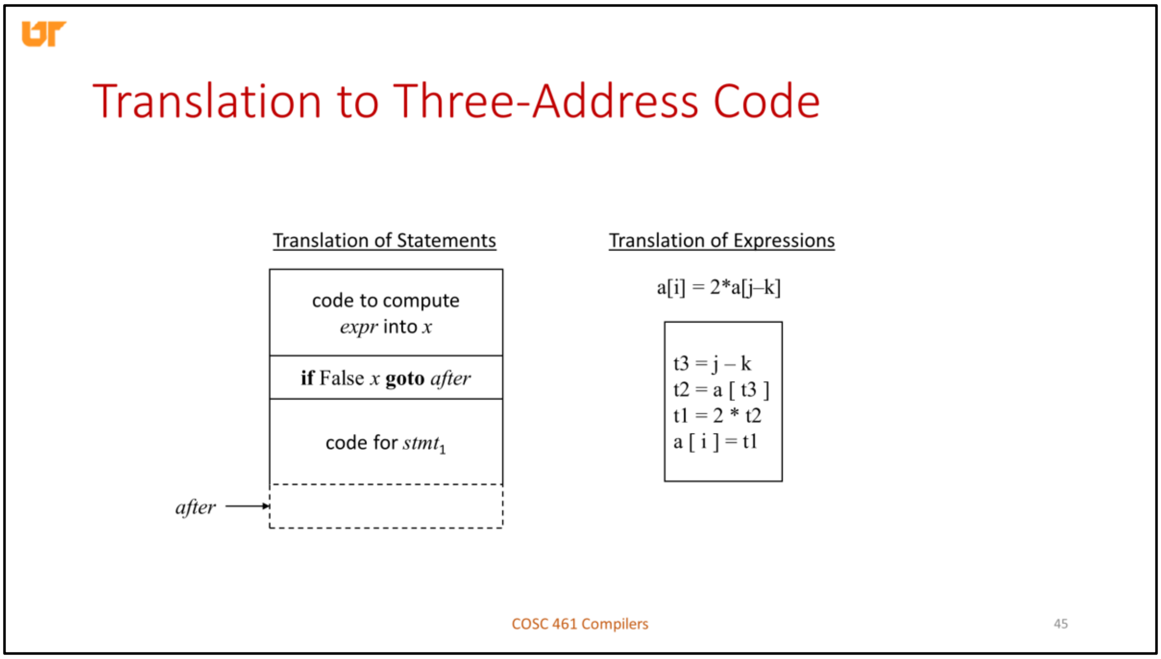
Where x, y, and z are names, constants, or compiler-generated temporary variables, and op is an operator. It's called three address code, because statements will have, at most, three variables.

We can have arrays – that look very much like an array you'd find in C code.

And we can have copies.

Also, there is control flow. The flow of instructions can be changed by a conditional or unconditional jump.

These allow us to implement loops and if statements.



Translation to Three-Address Code

Translation of Statements

Translation of Expressions

a[i] = 2*a[j–k]

```
t3 = j – k
t2 = a [ t3 ]
t1 = 2 * t2
a [ i ] = t1
```

Here we show examples of three-address codes for two types of constructs.

This one shows translation of an if statement:

if *expr* then *stmt*$_1$

So, first we have code to compute the expr to get some result x

Then, we can use the conditional branch in three-address code to test and branch on x. So, if x is true, we fall through to stmt$_1$, but if x is false, we branch over stmt$_1$ to the next instruction.

Here's another example of translating an array access from our source language. In this case, we are assigning to a[i]. So, since we can only have three variables in each statement, we get something like this:

t3 = j – k (to get the index)
t2 = a[t3] (to get the value of a[j-k])

t1 = 2 * t2 (gets the value of the RHS)
then assign a[i] = t1

That's it for chapter 2. I've given an overview of the front end. For the next part of the course, we'll do a deep dive into three separate parts of the front end – the lexical analyzer, the syntactic analyzer, and intermediate code generation.