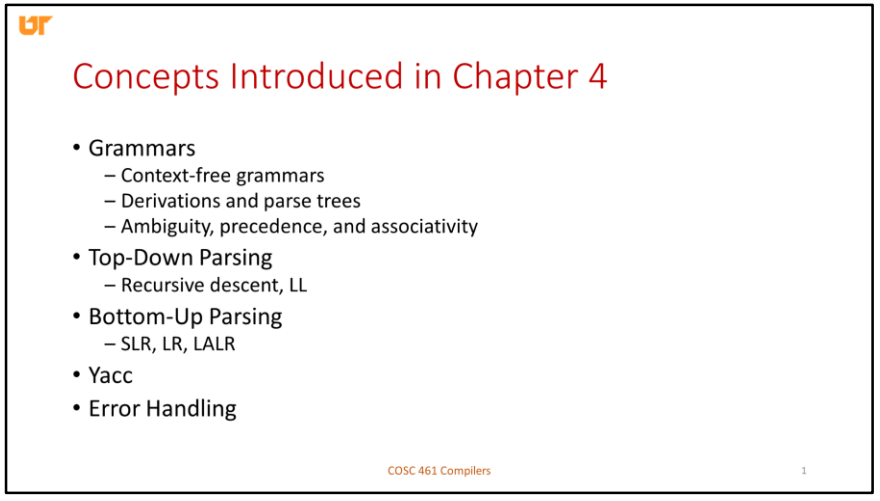
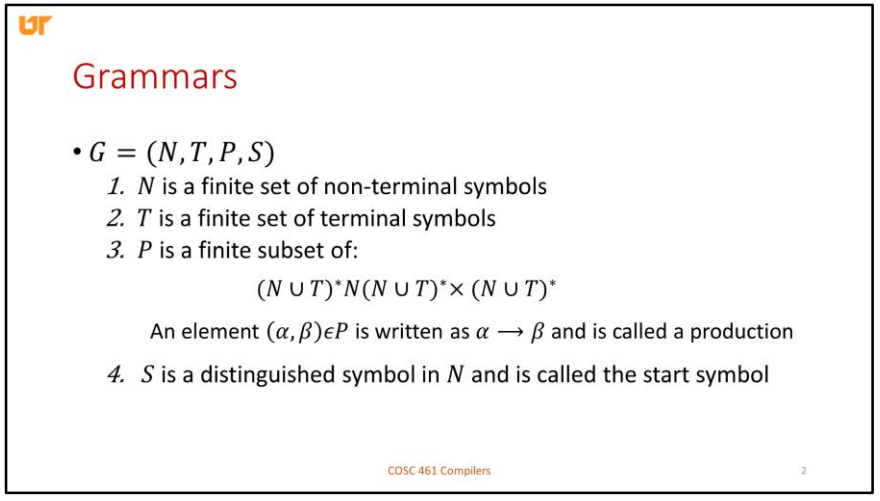


Again you will find that a lot of concepts covered in a formal languages or computer theory class are applied in both lexical and syntax analysis.

We will spend most of our time on top-down and bottom-up parsing.



You should recall from our previous classes that a grammar is used to define the syntax of a programming language. We can define grammars formally as a quadruple G = (N,T,P,S)

N is a finite set of non-terminals. Nonterminals are syntactic variables that denote sets of strings. For example, in previous grammars like:

expr ->expr + expr | factor
factor -> expr | id

expr and factor are non-terminals

T is a finite set of terminals. Terminals are just the legal tokens in a programming language. id is a terminal in the example grammar.

We also have productions. Here is the formal definition of productions.

This says the set of productions is a finite subset of the set denoted by this equation. The x in this equation is Cartesian product, which means all possible pairs. So, this says the productions are a finite subset of all the pairs where the left item in the pair has at least one non-terminal, and the right item is any combination of terminals and non-terminals.
Elements in the set of productions can be written with an arrow – which gives us the form we are used to seeing already. The main thing to note here is that productions always have a non-terminal on the LHS.

And finally, we have a non-terminal symbol distinguished as the start symbol for the grammar.

Example Grammar

$$
\begin{aligned}
expression &\rightarrow expression + term \\
expression &\rightarrow expression - term \\
expression &\rightarrow term
\end{aligned}
$$

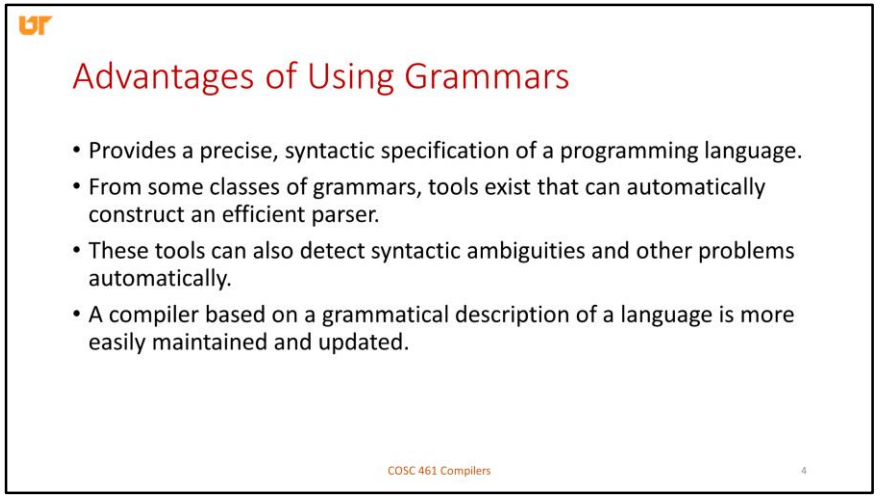
$$
\begin{aligned}
term &\rightarrow term * factor \\
term &\rightarrow term / factor \\
term &\rightarrow factor
\end{aligned}
$$

$$
\begin{aligned}
factor &\rightarrow (expression) \\
factor &\rightarrow \mathbf{id}
\end{aligned}
$$

Here is a grammar for simple arithmetic expressions. We'll use this grammar throughout the chapter for several examples.

Q. What are the non-terminals in this grammar?
A. expression, term, factor

Q. What are the terminals?
A. +,-,*,/,(,),id

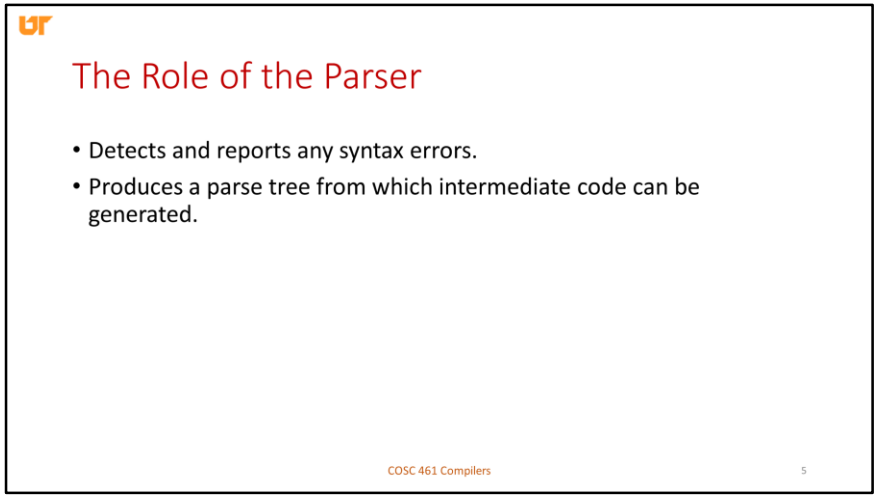


Grammars provide a precise syntactic specification of a programming language.

This is helpful for the compiler because it makes translation easier. For some classes of grammars, we can create tools to automatically construct an efficient parser directly from the grammar.

These tools can detect syntactic ambiguities and other problems with your language automatically.

Not only are grammars helpful for the compiler itself, but they can also be helpful to the programmer. A compiler based on a grammatical description is more easily maintained and updated. Grammars tend to be very concise, which makes thinking about them and working with them easier.



The parser has two main functions.

First, it checks the syntax of your code and reports if there are any syntactical errors in your program.

I should note that most of the errors that you see detected by your compiler are syntax errors that are found by the parser. Some of the other types of errors you'll find include: lexical errors (detected by the lexical analyzer) (these are not very common), semantic (more common), and logical (also more common, but not detected by a compiler).

Position of Parser in Compiler Model

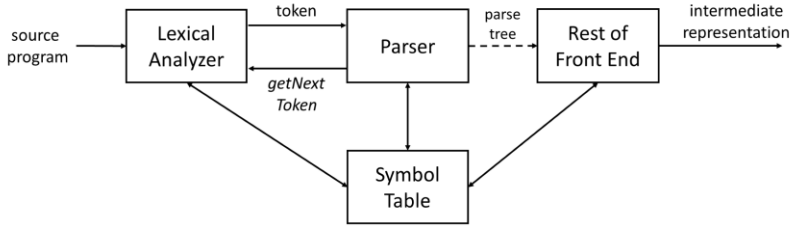


Figure 4.1: Position of parser in compiler model

Here is a figure showing the position of the parser in the compiler model. Notice that the parser interacts with both the lexical analyzer and symbol table It gets tokens from the lexical analyzer and inserts and reads from the symbol table. The symbol table will be used for doing semantic checks like type checking during translation. In most cases, it is actually the parser that makes calls to place identifiers into the symbol table.

Conventions for Specifying Grammars

- Terminals
  - Lowercase letters early in the alphabet, such as a, b, or c
  - Punctuation and operator symbols $[(,),',',+,-]$
  - Digits
  - Boldface words, such as **id** or **then**
- Non-terminals
  - Uppercase letters early in the alphabet, such as A, B, or C
  - S is the start symbol
  - Lowercase italic names, such as $expr$ or $stmt$

I want to go through some of the conventions we'll use throughout the slides for this chapter. It's important to remember these conventions, otherwise the slides and examples are not going to make a whole lot of sense.

Read the slide …

Conventions for Specifying Grammars (cont.)

- Grammar symbols (non-terminals or terminals)
  - Uppercase letters late in the alphabet, such as X, Y, or Z
- Strings of terminals
  - Lowercase letters late in the alphabet, such as $u, v, \ldots, z$
- Sentential form (string of grammar symbols)
  - Lowercase Greek letters, such as α, β, or γ

Read it …

Chomsky Hierarchy

- A grammar is said to be
  1. regular if it is
     a) right-linear: where each production P has the form
        $A \rightarrow wB$ or $A \rightarrow w$
     b) left-linear: where each production P has the form
        $A \rightarrow Bw$ or $A \rightarrow w$

        where $A, B \in N$ and $w \in T^*$

OK – now I want to talk about different types of grammars.

Q. How many people have heard of the Chomsky hierarchy?

This is a way of classifying languages based on their structure. There are four types of languages in the hierarchy

(draw up) regular <= context-free <= context-sensitive <= unrestricted

OK – so let's look at how each of these are defined.

A regular grammar is one that is right linear or left linear. A right linear grammar has productions of the form: A -> wB or A->w, where w is a string of terminals and B is a non-terminal. So, with right-linear grammars, we recurse down the right side during parsing.

Left-linear grammars are just the opposite. Their productions have the form A->Bw or A->w

Regular grammars are equivalent to regular expressions and regular languages.

Regular grammars can be recognized by a simple finite automata (FA).

Chomsky Hierarchy

2. context-free: if each production in P is of the form
   $A \rightarrow \alpha$ where $A \in N$ and $\alpha \in (N \cup T)^*$
3. context-sensitive: if each production in P is of the form
   $\alpha \rightarrow \beta$ where $|\alpha| \leq |\beta|$
4. unrestricted: if each production in P is of the form
   $\alpha \rightarrow \beta$ where $\alpha \neq \varepsilon$

Context-free grammars have productions with this form:

A->alpha, where A is a non-terminal and alpha is a string of non-terminals and terminals of any length.
Context free languages can be recognized by a pushdown automata (PDA), which is just like the finite automata we had already seen with an additional stack.

A context sensitive grammar is one that has productions like this:

Alpha->beta, alpha and beta are strings of non-terminals and terminals, but the length of alpha has to be less than or equal to beta.

Context sensitive languages can be recognized by a linear bounded automaton (LBA), which is just a turing machine with a finite amount of tape.

Unrestricted languages have almost no restrictions. It is just that alpha cannot be the empty string. These languages can be recognized by a turning machine.

Derivation

- Derivation: a sequence of replacements from the start symbol in a grammar by applying productions
- Example Grammar:

$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id}$$

- Derive $-(\mathbf{id} + \mathbf{id})$ from the grammar

$$E \Rightarrow -E \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$$

- Thus, E derives $-(\mathbf{id} + \mathbf{id})$ or $E \Rightarrow^{+} -(\mathbf{id} + \mathbf{id})$

Now let's talk about how you derive strings in the language from a grammar.

A derivation is a sequence of replacements from the start symbol in a grammar by applying productions

Each step in a derivation involves two choices:
1) which nonterminal to replace at that current point
2) which alternative to use for that nonterminal

So, let's do an example. Given this grammar (write up the grammar), I want to derive –(id+id) and see if its in the language.

So , to do that, starting from the start symbol, E:

The double arrow means we derive a single step. *=> and +=> mean derive in zero or more or derive in one or more steps. Since I could not find an easy way to put the + and * above the arrow – I'm writing it like this.

Derivation

- Leftmost Derivation
  - Each step replaces the leftmost nonterminal
  - Derive $\mathbf{id} + \mathbf{id} * \mathbf{id}$ using leftmost derivation

$$E \Rightarrow E + E \Rightarrow \mathbf{id} + E \Rightarrow \mathbf{id} + E * E \Rightarrow \mathbf{id} + \mathbf{id} * E \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$

- $L(G)$ – language generated by the grammar G
- Sentence of G
  - If $S +\Rightarrow w$, where w is a string of terminals in $L(G)$
- Sentential form
  - If $S *\Rightarrow \alpha$, where α may contain nonterminals

OK – leftmost derivation is a special kind of derivation where each step replaces the leftmost terminal.

If we want to derive id+id*id, here's the leftmost derivation.

We can also do rightmost derivation. This is where each step replaces the rightmost nonterminal. Let's work through this example with rightmost derivation.

We will see that leftmost derivations are used in top down parsers
and rightmost derivations are used in bottom up parsers.

OK – now we have a few more definitions. L(G) is just the language generated by the grammar G. A sentence of G is a string of terminal symbols in L(G) called w, where, starting at the start symbol of G, we can derive w in one or more steps from S.

Sentential form is similar. This is just a string alpha derived in zero or more steps from the start symbol of G, which may contain non-terminal symbols as well.

So, a sentence is just sentential form with only terminal symbols.

Parse Tree

- 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 has the properties:
  - The root is labeled by the start symbol
  - Each leaf is labeled by a token or ε
  - Each interior node is labeled by a nonterminal
  - If A is a nonterminal labeling some interior node and $X_1, X_2, X_3, \ldots, X_n$ are the labels of the children of that node from left to right, then $A \rightarrow X_1, X_2, X_3, \ldots, X_n$ is a production of the grammar.

A parse tree pictorially …

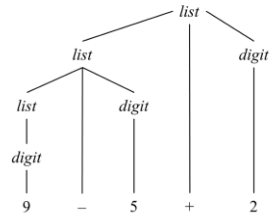
A parse tree is a graphical representation for a derivation.
Remember that a parse tree is also called a concrete syntax tree.

Given a context-free grammar, a parse tree has these properties: …

Parse trees can give you an idea of the syntactical structure, but it ignores the order in which symbols in the sentential form are replaced.

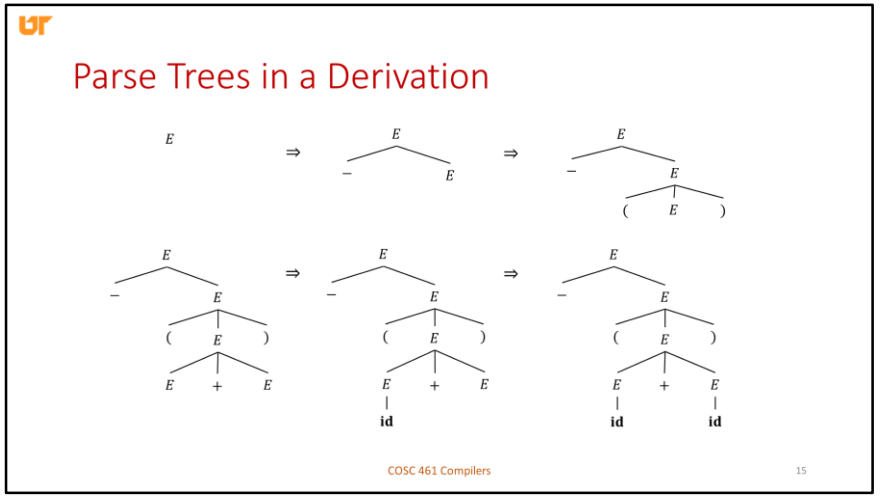
So, there might be several different possible parse trees depending on how you do the derivation. Many of these variations are eliminated if we consider only a leftmost or rightmost derivation.

Example of a Parse Tree



$list \rightarrow list + digit \mid list - digit \mid digit$

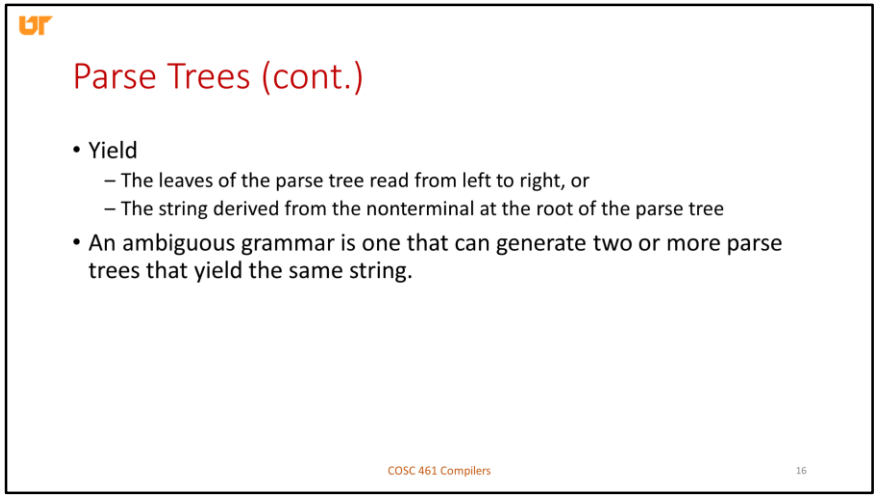
So this is a parse tree representing a derivation of a specific string in the language defined by the grammar.



Here's an example of parsing -(id+id) with that grammar.

So at each step we add to a leaf node that contains a nonterminal.

This example illustrates top down parsing since it builds the tree from the root towards the leaves.

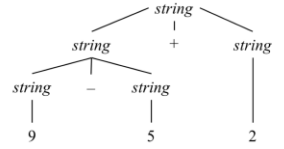


Read the slide

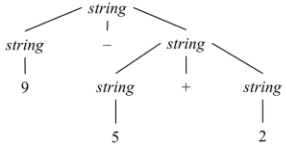
Ambiguous grammars cause problems since more than one parse or meaning can be interpreted for a given string in the language.

Example of an Ambiguous Grammar

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



a). $string \Rightarrow string + string \Rightarrow string - string + string$
$\Rightarrow 9 - string + string \Rightarrow 9 - 5 + string \Rightarrow 9 - 5 + 2$



b). $string \Rightarrow string - string \Rightarrow 9 - string$
$\Rightarrow 9 - string + string \Rightarrow 9 - 5 + string \Rightarrow 9 - 5 + 2$

It is also possible that one can produce more than one leftmost or rightmost derivations for the same sentence.

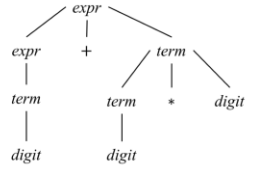
Both derivations in the example are leftmost derivations.

(a) implies that the subtraction is done first [result value is 6].
(b) implies that the addition is done first [result value is 2].

Precedence

- By convention, $*$ has higher precedence than $+$
- Example: $9 + 5 * 2$

$$expr \rightarrow expr + term \mid term$$
$$term \rightarrow term * digit \mid digit$$



Precedence determines the order that operands are applied to different operators. So, for example, by convention, * has higher precedence than +

In a grammar, we can establish precedence through the use of extra nonterminals. Nonterminals with productions defined at the lowest level will have the highest precedence.

In a parse tree, operators with higher precedence appear lower in the parse tree. So, in this example for 9-5*2, we structure our grammar so that the multiplication rule is matched after the addition rule. This will result in a parse tree that looks like this.

Let's see what will happen if the operators with different precedence are not separated.

Example grammar:
expr --> expr + digit | expr * digit | digit

Parse String: 9 + 5 * 2

The leftmost derivation will be:

expr => expr * digit
=> expr + digit * digit
=> digit + digit * digit
=> 9 + 5 * 2

Build the parse tree from this derivation. It will grouped as: (9+5)*2.

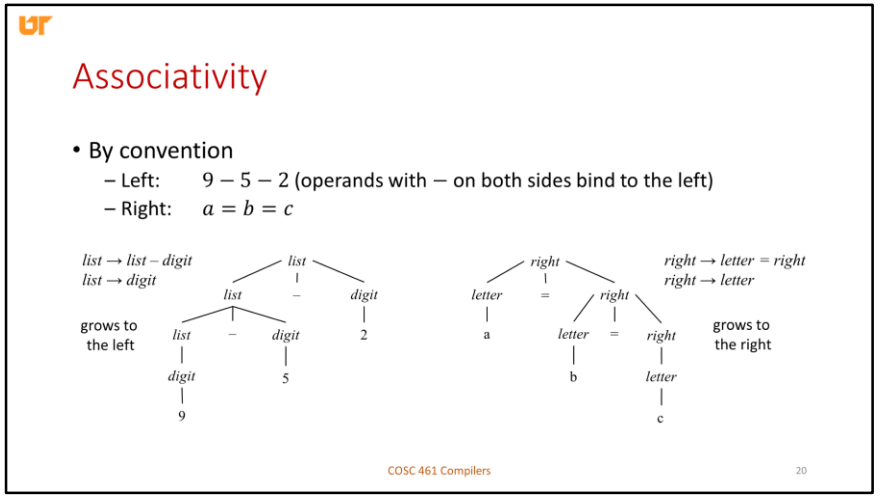
Precedence (cont.)

- If different operators have the same precedence then they are defined as alternative productions of the same nonterminal.

$$
\begin{aligned}
expr &\rightarrow expr + term \mid expr - term \mid term \\
term &\rightarrow term * factor \mid term / factor \mid factor \\
factor &\rightarrow digit \mid (expr)
\end{aligned}
$$

If different operators have …

So here + and - have the same precedence, and * and / have the same precedence.
() has the highest precedence since it is defined at the lowest level.



When an operand has an operator on the left and the right that have the same precedence, associativity indicates which operator will use the operand first.

Left associativity
Operand is taken by operator on its left (e.g. usually +, -, *, /).
We can establish left associativity by making the grammar productions left recursive.
Note the parse tree grows to the left.

Right associative
Operand is taken by operator on its right (e.g. = in C, ** in FORTRAN).
We can establish right associativity by making the grammar productions right recursive.
Note the parse tree grows to the right.

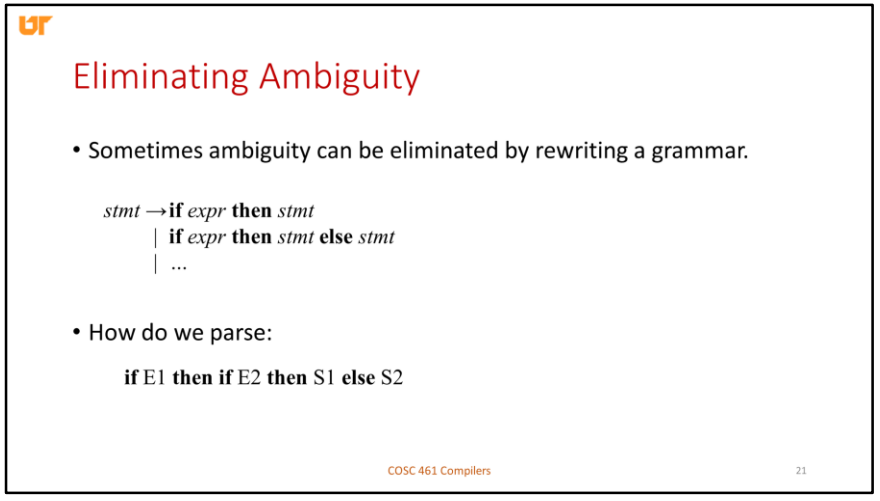
Show counterexample, with the example grammar:

expr → digit – expr | digit + expr | digit

If String is: 9 - 5 + 2 then, Leftmost derivation will be:

expr => digit - expr
=> 9 - expr
=> 9 - digit + expr
=> 9 - 5 + expr
=> 9 - 5 + digit
=> 9 - 5 + 2

Build the parse tree from this derivation. It will grouped as: 9-(5+2)



Sometimes we can eliminate ambiguity by rewriting a grammar.
We will see later that Yacc provides some other simple mechanisms for dealing with ambiguity.

Notice we can parse it as:

if E1 then (if E2 then S1) else S2

or

if E1 then (if E2 then S1 else S2)

Parse Trees for an Ambiguous Sentence

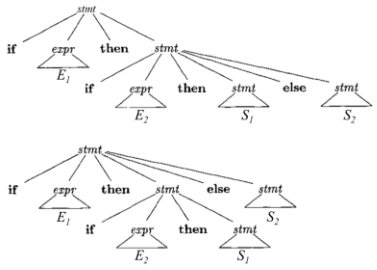
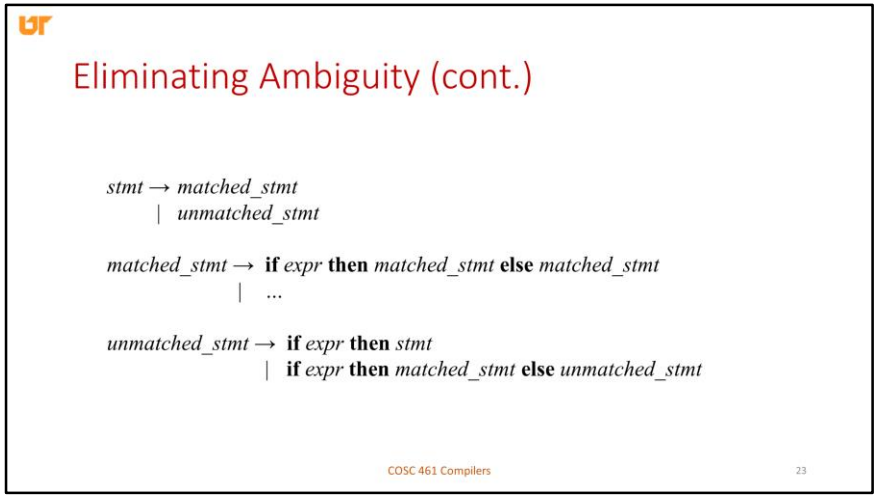


Figure 4.9: Two parse trees for an ambiguous sentence

For most programming languages, the first parse is desirable (the general rule is to match the else with the closest unmatched then).



This shows how we could re-write the grammar to only accept the first parse: if E1 then (if E2 then S1 else S2)

The idea with this grammar is that an if statement appearing between a **then** and an **else** must be matched (i.e. it must not end with a **then** followed by a stmt). Let's try a derivation for "if E1 then if E2 then S1 else S2" with the new grammar.

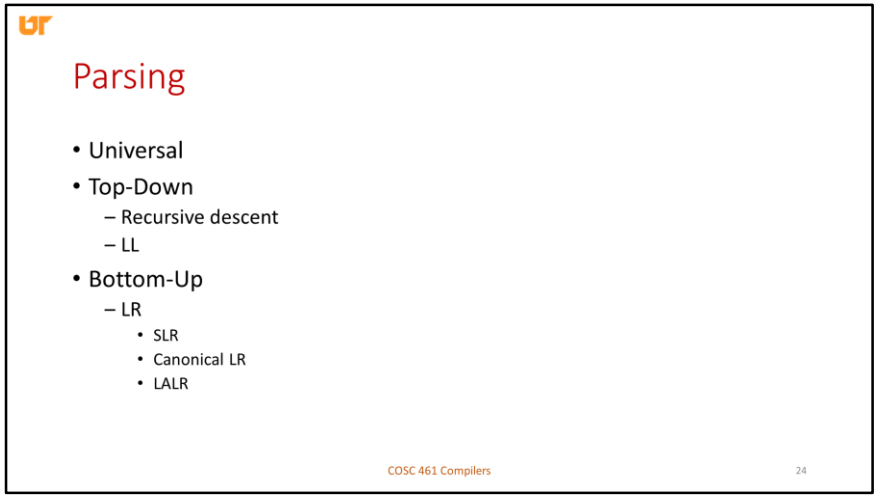
stmt => unmatched_stmt => if expr then stmt =>
if expr then matched_stmt => if expr then if expr then matched_stmt else matched_stmt

This is the only possible derivation.

If we try to expand the original stmt to a matched_stmt, we get:

if expr then matched_stmt else matched_stmt

Now the middle matched_stmt can only expand to:
if expr then if expr then stmt else stmt else stmt (which wont match our original statement).



Now, let's talk about parsing. We'll study several different parsing techniques. All of these techniques scan the input from left to right.

Universal

There are a couple universal parsing algorithms that can parse any context free grammar. These are the Cocke-Younger-Kasami algorithm or Earley's algorithm. They're both implemented using dynamic programming. But these are too inefficient to use in production compilers, complexity O(n^3).

Top-Down
We are going to study top-down parsers in a little more depth. These build a parse tree from the root to the leaves. These algorithms use a leftmost derivation to do the parse.

Recursive Descent is an example of a top-down parser. To implement a recursive descent parser, you associate a

procedure with each nonterminal in the grammar.
These parsers are the simplest to learn, and are also very efficient because they can use the run-time stack to implement the parser with recursion, and it doesn't have to dereference states from a table.

Another top-down method is called LL. This is a non-recursive, table-driven approach.
The first ``L'' means that we scan the input from left to right.
The second ``L'' means that this kind of parser produces a leftmost derivation.

If you see LL(1), that means the parser only uses one symbol of lookahead to make a decision.
Another name for an LL(1) parser is predictive parsing since the parsing table has no multiply defined entries and backtracking is not needed.

And we'll also look at bottom-up parsers.
A bottom-up parser starts from the leaves and works up to the root. #It also scans from left to right, but it constructs a rightmost derivation in reverse order.

#LR parsing is a bottom-up approach and it can parse a large class of context-free grammars.
An LR parser uses a DPDA (deterministic push down automata) to parse the input program.
``L'' indicates that it is performing a left-to-right scan of the input.
``R'' indicates that it constructs a rightmost derivation in reverse.
And again, if we have an LR(1) parser, that means we use one symbol of lookahead to implement the parser.

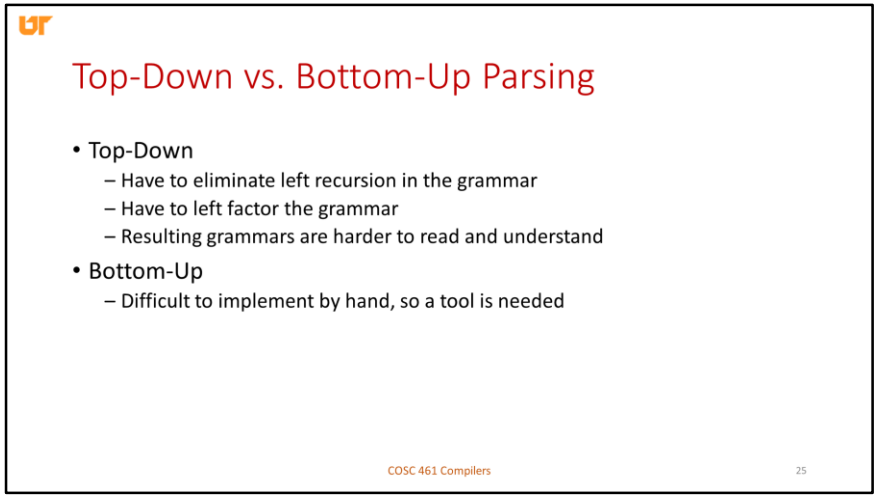
Bottom up parsers can handle a larger class of languages than using a predictive parser.
However, the main drawback to these approaches is that they're difficult to implement. If you want to use an LR(1) parser, you usually have to use a parser-generator tool, like yacc, to implement your parser.

So, there are a few types of LR parser that we'll discuss.
An SLR parser is a "simple" LR parser.

This is the easiest of the LR techniques to implement, but it is also the least powerful (i.e. it accepts the smallest class of languages).

We also have the canonical LR parser.
This is the most powerful of the LR parsers, but it is also the most expensive (by that I mean, it will probably be slower than the other types of LR parsers and require more memory).

And we also have LALR parsers.
LALR stands for a Lookahead LR parser.
In terms of power and cost, this one is in between SLR and canonical LR.
This is the approach that is used in most parser generators, and is the approach that's used by yacc.



This slide indicates some of the tradeoffs between top-down and bottom-up parsers.

We cannot use a top-down parser unless the grammar is in the proper format. So, to implement one, we need to eliminate left recursion in the grammar. That is, if we have a production like this:

A->Aa

Where we recurse down the left side, then the top-down parser will go into an infinite loop. So, we have to transform the grammar somehow to remove that.

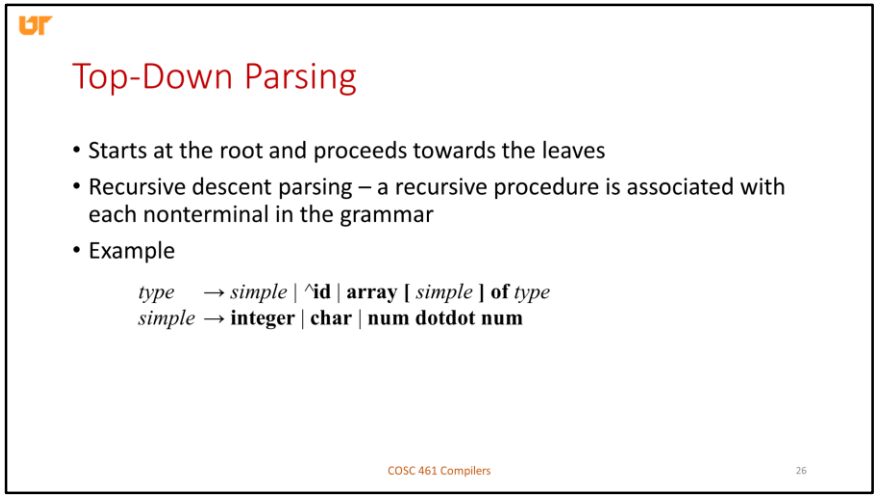
We might also need to left factor the grammar. This is necessary when we have two productions, and the choice between the two productions is not clear:

stmt → if expr then stmt else stmt | if expr then stmt

On seeing the input if, it's not clear which production to choose. So, we have to transform the grammar to get rid of these situations.

One downside of this is that the resulting grammars are much harder to read and understand than the original grammar.

Bottom-up approaches are nice because they accept a larger class of grammars. We don't have to do these transformations, but, as I mentioned previously, bottom-up parsing is difficult to implement by hand, so a tool is needed.



Top-down parsing starts at the root of your parse tree and proceeds toward the leaves. Top-down parsing is an attempt to find a leftmost derivation for an input string.

Recursive descent parsing is a type of top-down parsing that associates a recursive procedure with each non-terminal in the grammar.

We'll go over an example of how to implement a recursive descent parser for this grammar in the next few slides (write it up).

Top-Down Parse Example

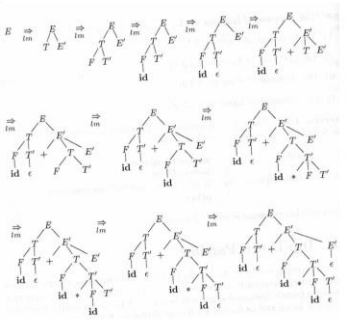
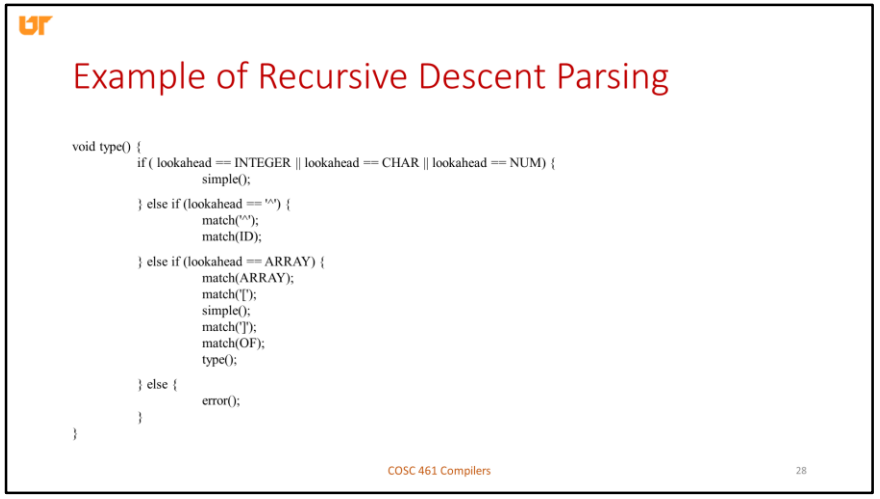


Figure 4.12: Top-down parse for **id** + **id** * **id**

First, here's a figure from the book that shows a leftmost derivation for the string id+id*id.

Notice that, at each point, the leftmost nonterminal is replaced with some production of that nonterminal.

We will see how the algorithm selects which production rule to apply later.



Here is an implementation for a recursive descent parser for the grammar.

Can see in the example that there is:
1. A procedure for each nonterminal. We determine which production to use by checking the lookahead symbol.
2. Each procedure for each nonterminal checks the form of the RHS of the production.

Look at the next slide to show definitions of simple and match.

So, there are a couple things we need to make sure of before we write a recursive descent parser for our grammar.

If the lookahead at the start of the RHS for the set of productions is not unique, then the grammar needs to be augmented by left factoring the grammar. This will basically just make it so productions start with unique symbols.

Also, the grammar cannot be left recursive.

Q. What would happen if the grammar was left recursive?
A. The recursive descent parser would go into an infinite loop.

If it's not left recursive, we'll always consume at least one input symbol with the match function.

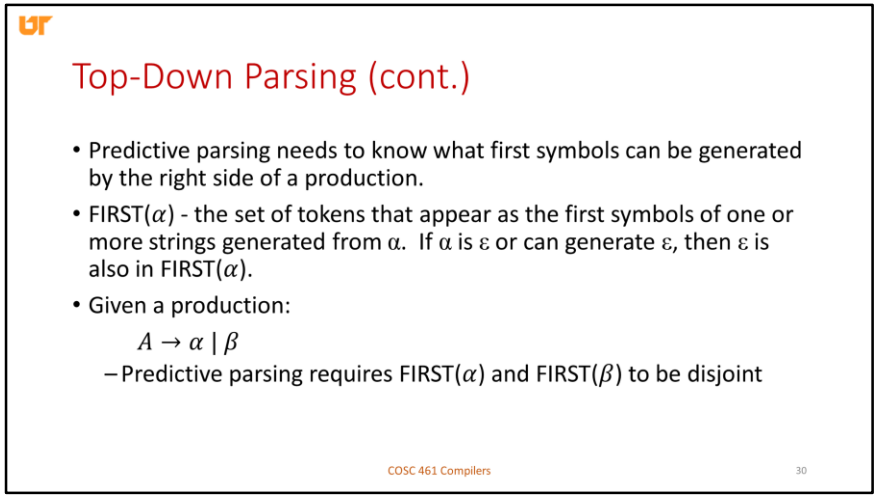
Example of Recursive Descent Parsing (cont.)

```
void simple() {
        if (lookahead == INTEGER) {
                match(INTEGER);
        } else if (lookahead == CHAR) {
                match(CHAR);
        } else if (lookahead== NUM) {
                match(NUM);
                match(DOTDOT);
                match(NUM);
        } else {
                error();
        }
}
```

```
void match(token t) {
        if (lookahead == t) {
                lookahead = nexttoken();
        } else {
                error();
        }
}
```


Show how simple implementation matches the grammar.

The match function advances the lookahead symbol.



Predictive parsing needs to know …

The formal definition for predictive parsing is that if a nonterminal A is to be expanded and there are productions:

A -> $alpha_1$ | $alpha_2$ | ... | $alpha_n$

and the next input symbol is a, then we can uniquely choose the appropriate $alpha_i$ by examining the input symbol (i.e. no backtracking is required).

Note, that a predictive parser can be recursive or nonrecursive.

Now, What if an $alpha_i$ begins with a nonterminal B?
In that case, it may not be immediately apparent what symbols can be the first terminals in $alpha_i$

For that reason, we might need to compute FIRST($alpha_i$). This is the set of tokens that appear as the first symbols of one or more strings generated from alpha. If alpha is epsilon or can generate epsilon, then epsilon is also in FIRST(alpha).

So, for predictive parsing to work, if we're given a production of the form A -> alpha | beta, we need to be able to ensure that FIRST(alpha) and FIRST(beta) are disjoint, that is, they don't derive any of the same tokens.

We'll talk more about computing FIRST(alpha) a little later.

Eliminating Left Recursion

- Recursive descent parsing loops forever on left recursion
- Immediate left recursion
  - Replace $A \rightarrow A\alpha \mid \beta$ with $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon$
- Example:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \mathbf{id}$

A	α	β
E	+T	T
T	*F	F

becomes:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
…

So, as we saw recursive descent parsing will loop forever if we have left recursion in our grammar. So, we need to eliminate it.

Here is the simple rule for eliminating immediate left recursion.
We replace productions of the form A -> A alpha | beta with A -> beta A', A' -> alpha A' | epsilon

That's it – this all you need for eliminating most examples of left recursion.
Why does this rule for eliminating immediate left recursion work?

Notice that with this grammar:

A -> A alpha | beta

Let's expand this out a few times:

We have A => A alpha => A alpha alpha => A alpha alpha … => beta alpha alpha …

Eventually, we take the A -> beta production to end the recursion. So, as a regular expression, this would be:

beta alpha*

Which looks like what are revised form will generate. Also, note that the revised form is still recursive, but crucially, it's not left-recursive, which is the requirement for implementing a top-down parser.

Here's another example of applying the rule:

Notice that E -> E+T | T and T -> T*F | F are both left recursive. So, to eliminate them, we just find what are A, alpha, and beta in each production and apply the immediate left recursion rule to eliminate the left recursion.

Q. Now, what do we replace T->T*F | F with?
A. T -> FT', T' -> *FT' | epsilon

Q. What about the F productions?
A. No change is needed since it is not left recursive.

What if the grammar is not immediately left recursive? That is, what if:

A +=> A alpha, but the productions are not in the immediate left recursion form?

For instance:

A -> B $alpha_1$ | $alpha_4$

B -> C $alpha_2$
C -> A $alpha_3$

This grammar is left recursive because A => B $alpha_1$ => C $alpha_2$ $alpha_1$ => A $alpha_3$ $alpha_2$ $alpha_1$

Eliminating Left Recursion (cont.)

Given a set of nonterminals in order $A_1, A_2, \ldots, A_n$
for i=1 to *n* do {
    for j=1 to i–1 do {
        replace each production of the form $A_i \rightarrow A_j\gamma$
        with productions $A_i \rightarrow \delta_1\gamma \mid \ldots \mid \delta_k\gamma$ where
        $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are the current A_j productions
    }
    eliminate the immediate left recursion in the A_i productions
    eliminate ε transitions in the A_i productions
}

This fails only if there are cycles (i.e. $A \Rightarrow^+ A$) or $A \rightarrow \varepsilon$ for some A.

So, to eliminate left recursion in the general case, we have this algorithm.

This says given a set of nonterminals in the order A_1 A_2 … A_n

For each i=1 to n
And then for each j=1 to i-1

So, consider that A_i is the current non-terminal that's on the LHS of the production we're considering. And A_j is a non-terminal that comes before A_i in the order we were given. If we have a production of this form:

A_i -> A_j gamma

We might have an instance of left recursion that's not immediate. So, to avoid that, we get rid of productions of that form, and we replace them with productions that replace A_j with the RHS of the A_j productions, like this:

A_i -> delta 1 gamma | delta 2 gamma | … | $delta_k$ gamma, where A_j -> $delta_1$ | $delta_2$ | … | $delta_k$ are the current productions of A_j

Then, all we have left should be immediate left recursion, so we just eliminate the immediate left recursion in each A_i using the rule from the previous slide.

And, we eliminate epsilon transitions in the A_i productions. For example, to eliminate epsilon productions, if we have productions of the form:

B -> $alpha_1$ C $alpha_2$
C -> epsilon | $alpha_3$ C $alpha_4$ | …
=>
B -> $alpha_1$ C $alpha_2$ | $alpha_1$ $alpha_2$
C -> $alpha_3$ C $alpha_4$ | $alpha_3$ $alpha_4$

In other words, we just add a production where the epsilon symbol is replaced by the empty string.

I should note, the only time this algorithm will fail is if there are cycles in the grammar, i.e. A eventually derives itself, or A -> epsilon, for some nonterminal A.

OK, now I want to step through an example of applying this algorithm. Say, we have this grammar:

$X \rightarrow YZ \mid a$
$Y \rightarrow ZX \mid Xb$

$Z \rightarrow XY \mid ZZ \mid a$

Note that:
X -> YZ, Y -> ZX, Z -> XY (i.e. X => YZ => ZXZ => XYXZ)
So there is some left recursion.

Let's let A1 == X, A2 == Y, A3 = Z.
Now, let's work through the algorithm:

At i = 1, we never even enter the inner loop, so we just eliminate immediate left recursion. There is no immediate left recursion, so there's nothing to do.

Next, we have:
i=2, j=1

We check if we have any productions of the form: Y -> X gamma
We do, we have Y->Xb, so we replace it with

Y -> YZb | ab | ZX

And eliminate immediate left recursion (write up the rule for left recursion):
Y -> ZXY' | abY'
Y' -> ZbY' | epsilon

Eliminate epsilon transitions:
Y -> ZXY' | abY' | ZX | ab
Y' -> ZbY' | Zb

Next, we have, i=3, j=1

Do, we have anything of the form Z -> X gamma? We do, we have:
Z -> XY
So, we replace that with:

Z -> YZY | aY

So, now i=3, j=2
Anything of the form Z -> Y gamma? Yes, so replace it:

Z -> ZXY'ZY | abY'ZY | ZXZY | abZY | aY | ZZ | a

Eliminate immediate left recursion:

Z→ abY'ZYZ' | abZYZ' | aYZ' | aZ'
Z'→ XY'ZYZ' | XZYZ' | ZZ' | ε

eliminate ε transitions
Z→ abY'ZYZ' | abY'ZY | abZYZ' | abZY | aYZ' | aY | aZ' | a
Z'→ XY'ZYZ' | XY'ZY | XZYZ' | XZY | ZZ' | Z

Now, the final grammar is:

X -> YZ | a
Y -> ZXY' | abY' | ZX | ab
Y' -> ZbY' | Zb
Z→ abY'ZYZ' | abY'ZY | abZYZ' | abZY | aYZ' | aY | aZ' | a

Z'→ XY'ZYZ' | XY'ZY | XZYZ' | XZY | ZZ' | Z

Example of Eliminating Left Recursion

$X \rightarrow YZ \mid a$
$Y \rightarrow ZX \mid Xb$
$Z \rightarrow XY \mid ZZ \mid a$

$A_1 = X \qquad A_2 = Y \qquad A_3 = Z$
i = 1 (eliminate immediate left recursion)
nothing to do

Example of Eliminating Left Recursion (cont.)

i = 2, j = 1

Y → Xb ⇒ Y → ZX | YZb | ab

now eliminate immediate left recursion

Y → ZXY' | abY'

Y' → ZbY' | ε

now eliminate ε transitions

Y → ZXY' | abY' | ZX | ab

Y' → ZbY' | Zb

i = 3, j = 1

Z → XY ⇒ Z →YZY | aY | ZZ | a

Example of Eliminating Left Recursion (cont.)

i = 3, j = 2

$Z \rightarrow YZY \Rightarrow Z \rightarrow ZXY'ZY \mid ZXZY \mid abY'ZY \mid abZY \mid aY \mid ZZ \mid a$

now eliminate immediate left recursion

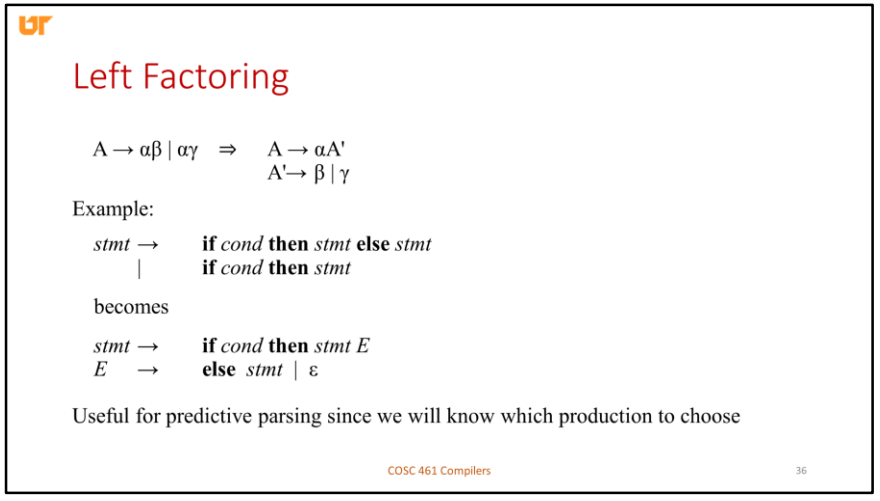
$Z \rightarrow abY'ZYZ' \mid abZYZ' \mid aYZ' \mid aZ'$

$Z' \rightarrow XY'ZYZ' \mid XZYZ' \mid ZZ' \mid \varepsilon$

eliminate ε transitions

$Z \rightarrow abY'ZYZ' \mid abY'ZY \mid abZYZ' \mid abZY \mid aY \mid aYZ' \mid aZ' \mid a$

$Z' \rightarrow XY'ZYZ' \mid XY'ZY \mid XZYZ' \mid XZY \mid ZZ' \mid Z$



OK – here's the approach for left factoring a grammar.

Left factoring is a transformation that is useful for building predictive parsers. In these parsers, you only have one symbol of lookahead, and if the grammar is written a certain way it might not always be clear which production should be selected.

If we have something of the form:

A -> alpha beta | alpha gamma
=>
A -> alpha A'
A' -> beta | gamma

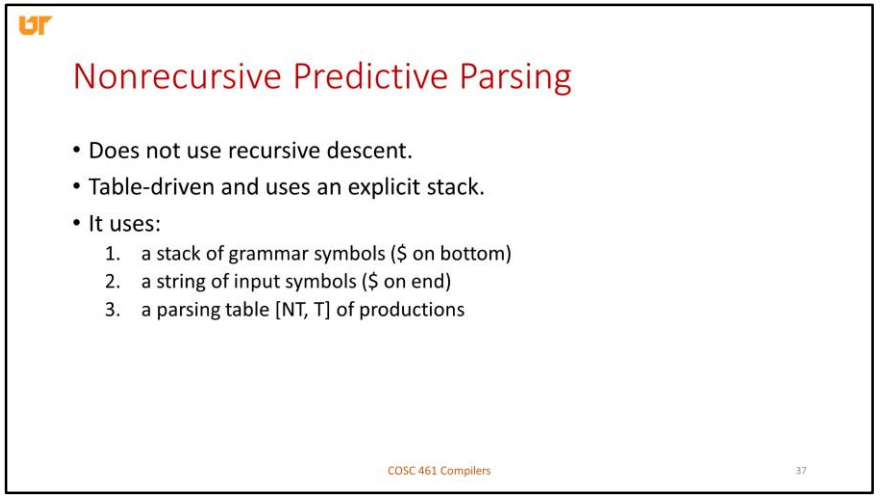
This example:

stmt -> **if** cond **then** else **stmt** | **if** *cond* **then** *stmt*
Becomes
stmt -> **if** *cond* **then** *stmt*
E -> **else** stmt | epsilon

Notice that, in this example, gamma is epsilon.

This approach allows the lookahead for a predictive parser to uniquely match one production. So, we don't have to do any backtracking.

This left factoring technique can be applied after eliminating left recursion.
This is probably the easiest way to do this because left factoring will never introduce left recursion.



Using recursive procedure calls to implement a stack abstraction might not be particularly efficient. So, we can also implement an explicit stack-based approach. We'll call this non-recursive predictive parsing.

This should also be more easily maintained when we change the grammar.

Non-recursive parsers are table driven and maintain an explicit stack as opposed to an implicit stack that is updated via recursive calls.

It uses …

Note for the parsing table, the T includes the $ symbol, which indicates (EOF).
$ on the stack means we're at the bottom of the stack.

Model of a Table-Driven Predictive Parser

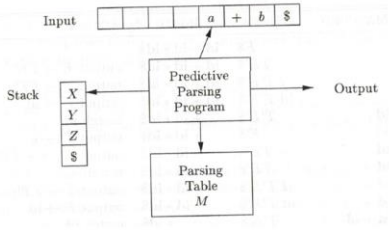


Figure 4.19: Model of a table-driven predictive parser

Here is a model of a table-driven predictive parser.

This algorithm explicitly builds a stack as opposed to building it through recursion. The $ symbol indicates the bottom of the stack and the end of the input.

The parser also will produce a leftmost derivation.

It works by looking at the symbol on the top of the stack and then looking at the current input symbol. If X is a nonterminal, it uses the non-terminal X along with the current input symbol to index into the parsing table M to determine the action to take.
Otherwise, it will check if X and 'a' are a match, in which case it will consume a, and increment the input pointer.

Algorithm for Nonrecursive Predictive Parsing

```
let a be the first symbol of w;
let X be the top stack symbol;
while ( X ≠ a ) {    /* stack is not empty */
        if ( X = a ) pop the stack and let a be the next symbol of w
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1Y2 ... Yk ) {
                output the production X → Y1Y2 ... Yk
                pop the stack;
                push Y1Y2 ... Yk onto the stack, with Y1 on top;
        }
        let X be the top stack symbol;
}
```


The input *w* is the string of terminals (received from the lexical analyzer) to be parsed.
Initially the stack contains a $ symbol and an S, where $ is on the bottom and S is the start symbol.

The algorithm says …

While the stack is not empty:
if the top of the stack matches the first input symbol a, then pop the stack and let a be the next symbol of w
else if (X is any other terminal or the table entry for M[X,a] is an error) then we error out
else if (M[X,a] is some production X->Y1Y2 … then
output the production, pop the stack, and push Y1, Y2, up to Yk onto the stack, but do it in reverse order
"replace top with rhs of production", but do it in reverse order.

Do an example, look at the next slide and draw it up, then work through the algorithm

Work through the example by hand.

Table starts with the headings:

Matched	Stack	Input	Action
	E$	id+id*id	
	TE'$	id+id*id	output E->TE'
	FT'E$	id+id*id	output T->FT'
	idT'E'$	id+id*id$	output F->id
Id	T'E'$	+id*id$	match id
…			

The derivation is leftmost:
E => TE' => FT'E' => idT'E' =>... => id+id*id

Parsing Table Example

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Figure 4.17: Parsing table M for Example 4.32

Draw this up if you do this example

Predictive Parser Example

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\, T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
id	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match id
id	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
id	$+\, TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \to + TE'$
id +	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match +
id +	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
id +	$\mathbf{id}\, T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
id + id	$T'E'\$$	$* \mathbf{id}\$$	match id
id + id	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
id + id *	$FT'E'\$$	$\mathbf{id}\$$	match *
id + id *	$\mathbf{id}\, T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
id + id * id	$T'E'\$$	$\$$	match id
id + id * id	$E'\$$	$\$$	output $T' \to \epsilon$
id + id * id	$\$$	$\$$	output $E' \to \epsilon$

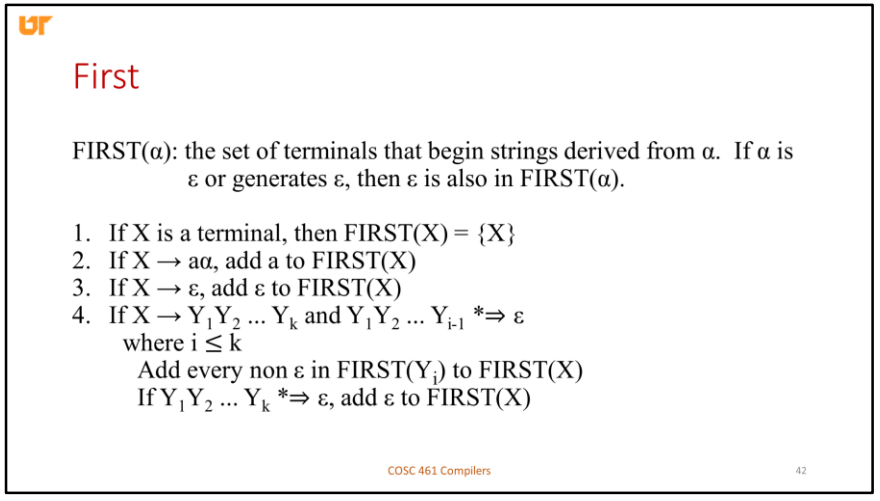
Figure 4.21: Moves made by a predictive parser on input **id + id * id**

[Leave algorithm for last slide on the screen.]
[Leave Figure 4.17 on screen.]
Work through the example by hand.

Table starts with the headings:

Matched	Stack	Input	Action
	E$	id+id*id	
	TE'$	id+id*id	output E->TE'
	FT'E$	id+id*id	output T->FT'
	idT'E'$	id+id*id$	output F->id
Id	T'E'$	+id*id$	match id
…			

The derivation is leftmost:
E => TE' => FT'E' => idT'E' =>... => id+id*id



So, that algorithm shows how a nonrecursive predictive parser works. But notice, you first need to construct a parsing table.

To construct the parsing table you use with this approach, you need to be able to calculate FIRST and FOLLOW functions for your grammar.

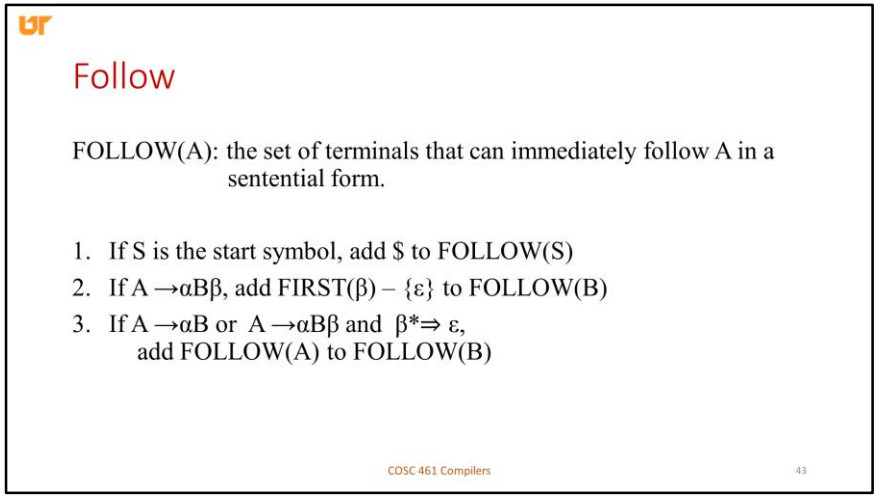
I mentioned this first function briefly a few lectures ago. FIRST(alpha) is the set of terminals …

Here is the definition of FIRST(alpha)

If X is a terminal, then FIRST(X) = {X}
If X -> a alpha, add a to FIRST(X)
If X -> epsilon, add epsilon to FIRST(X)
If X -> (go to some non-terminals) Y1, Y2, … , Yk and Y1, Y2, … Yi-1 derive the empty string, then where i <= k, add

every non epsilon in FIRST(Yi) to FIRST(X). If Y1, Y2, …, Yk derive epsilon, then add epsilon to FIRST(X)

The fourth rule says, if X goes to a non-terminal Y, add FIRST(Y) to FIRST(X). And if Y1 derives epsilon, then we also have to add FIRST(Y2), and so on. If Y1 through Yk derive epsilon, then we also have to add epsilon to FIRST(X).



After we calculate FIRST, we can calculate FOLLOW.

FOLLOW(alpha) is the set of terminals that can immediately follow A in a sentential form.

Here is the formal defintion of FOLLOW:

If S is the start symbol, add $ to FOLLOW(S)

If A -> alpha B beta, add FIRST(beta) – {epsilon} to FOLLOW(B)
Second rule says if beta follows a non-terminal B within a production – add FIRST(beta) to FOLLOW(B)

If A -> alpha B or A -> alpha B beta and beta derives epsilon, then everything in FOLLOW(A) is added to FOLLOW(B)
Third rule says that if a non-terminal B comes at the end of a production, or whatever follows B in the production derives epsilon, then add FOLLOW(A) to FOLLOW(B)

Example of Calculating FIRST and FOLLOW

Production	FIRST	FOLLOW
E → TE'	{ (, id }	{), $ }
E' → +TE' | ε	{ +, ε }	{), $ }
T → FT'	{ (, id }	{ +,), $ }
T' → *FT' | ε	{*, ε }	{ +,), $ }
F → (E) | id	{ (, id }	{*, +,), $ }

[Write productions on the board.
Calculate FIRST and FOLLOW from the productions.]

For the first production, we can see that E -> T, T -> F, and F goes to a couple terminals '(', and id. We can go ahead and add those to FIRST(F), and then, because we reach F immediately from T, and T immediately from E, we add those symbols to E and T as well.

For E', we reach '+' immediately, there's no non-terminal to consider, but we can also reach epsilon with this non-terminal. Same with T'.

By the first rule, '$' is in FOLLOW(E), and by the second rule, we add ')' to FOLLOW(E).
Also, notice that E' is only reached at the end of the E productions. Thus, FOLLOW(E') must be the same as FOLLOW(E) by the third rule.

For FOLLOW(T), T is reached by E -> TE' and FIRST(E') – {epsilon} is {'+'}, so we add that to FOLLOW(T) by rule 2. Also, E' derives epsilon, so FOLLOW(E) is added to FOLLOW(T) (by the third rule).

Then, for T', we add FOLLOW(T) to FOLLOW(T') by the third rule.

And for F, since F is followed by T' in T' -> *FT' | epsilon, by the second rule FIRST(T') – {epsilon} is in FOLLOW(F) (which adds the '*'), and since T' also => epsilon, we add FOLLOW(T') to FOLLOW(F).

We keep applying rules until nothing else can be added to the FIRST and FOLLOW sets.

Example of Calculating FIRST and FOLLOW

Production	FIRST	FOLLOW
X → Ya	{ }	{ }
Y → ZW	{ }	{ }
W→ c | ε	{ }	{ }
Z → a | bZ	{ }	{ }

[Have the students work through this example.]
FIRST(X) = {a, b} (X=>Y=>Z=>a|b)
FIRST(Y) = {a, b} (Y=>Z=>a|b)
FIRST(W) = {c, epsilon} (W => c|epsilon)
FIRST(Z) = {a, b} (Z=>a|b)

FOLLOW(X) = {$} (X is not reachable other than by the start symbol)
FOLLOW(Y) = {a} (because X -> Ya)
FOLLOW(W) = {a} (because Y -> ZW, add FOLLOW(Y) to FOLLOW(W) by third rule)
FOLLOW(Z) = {a, c} FIRST(W) – epsilon is in FOLLOW(Z) because Y -> ZW (second rule), FOLLOW(Y) is in FOLLOW(Z) because Y -> ZW and W => epsilon (third rule), and

Production First Follow

X -> Ya	{ a, b }	$
Y -> ZW	{ a, b }	{ a }
W -> c | €	{ c, € }	{ a }
Z -> a | bZ |	{ a, b }	{ a, c }

[Change the last set of production rules to: Z --> a | bZ | E,

Production	First	Follow
X -> Ya	{ a, b, c }	$
Y -> ZW	{ a, b, c, E }	{ a }
W -> c | E	{ c, E }	{ a }
Z -> a | bZ |E	{ a, b ,E}	{ a, c }

OR, What if instead of Y --> ZW, we have the production, Y --> WZ?}

Production	First	Follow
X -> Ya	{ a, b, c }	$
Y -> WZ	{ a, b, c }	{ a }
W -> c | E	{ c, E }	{ a, b }
Z -> a | bZ	{ a, b }	{ a }

Constructing Predictive Parsing Tables

For each $A \rightarrow \alpha$ do

1. Add $A \rightarrow \alpha$ to M[A, a] for each a in FIRST(α)
2. If ε is in FIRST(α)
   a) Add $A \rightarrow \alpha$ to M[A, b] for each b in FOLLOW(A)
   b) If $ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, $]
3. Make each undefined entry of M an error.

Here is the algorithm for constructing a predictive parsing table:

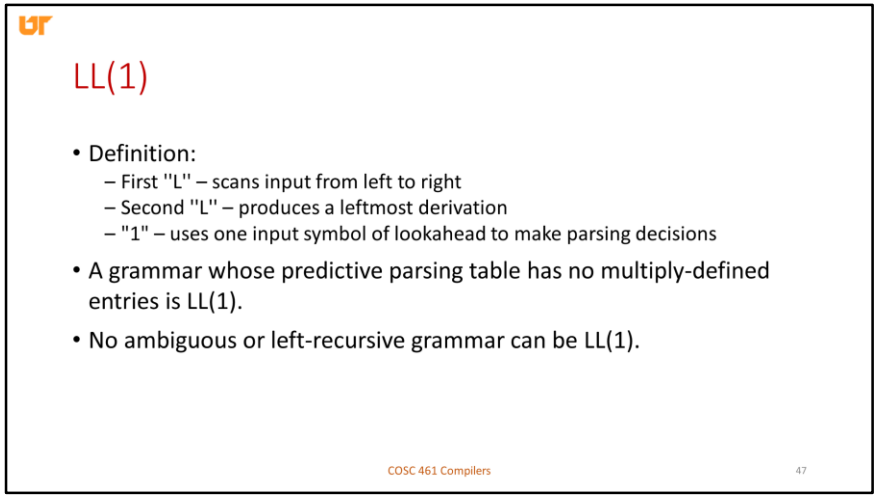
Note that the production A -> alpha indicates that alpha is just one rhs

So, for each production, Add A -> alpha to M[A,a] for each a in FIRST(alpha)
If E is in FIRST(alpha):
then add A->alpha to M[A, b] for each b in FOLLOW(A), or if $ is in FOLLOW(A), add A->alpha to M[A,$]
Note that we use FOLLOW(A) for A -> alpha when epsilon in the FIRST(alpha) since we have to pop the grammar symbol to get to whatever follows A.

And then, make each undefined entry of M an error.

Consider E->TE'. We look at FIRST(TE') = FIRST(T) = {(,id}, so we add E->TE' to (and id.

[Working from the list of productions of the first example of FIRST and FOLLOW,
construct the predictive parsing table.
Show Figure 4.17 again to check.]



We can construct predictive parsers with no backtracking required on a class of grammars called LL(1)

Read the definitions

You can determine if a grammar is LL(1) by constructing its parsing table. If the parsing table has no multiply-defined entries, then it's LL(1).
Note that ambiguous or left-recursive grammars cannot be LL(1).

Below is an example of what left recursion would produce in a predictive parsing table.

E -> E + a | a

FIRST(E+a)=a FIRST(a)=a

There is no epsilon, so

	+	a	$
	--	--	--
E		E -> E+a	
		E -> a	

You might think – oh maybe we can get around that by just always using one of the productions. Assume we always use the first production.

Stack	input
E	a+a
E+a	a+a
E+a+a	a+a

This will proceed until the stack overflows, just like in a recursive descent parser.

When is a Grammar LL(1)?

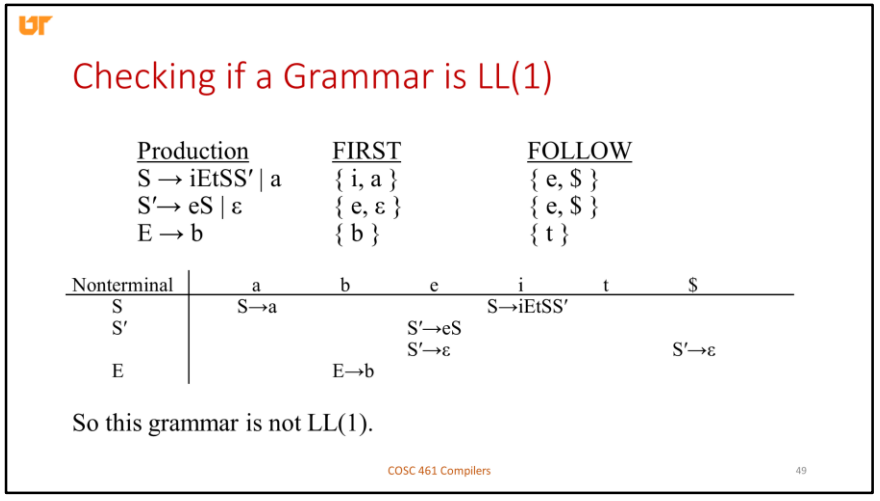
- A grammar is LL(1) iff whenever $A \rightarrow \alpha \mid \beta$, then the following conditions hold:
  1. For no terminal a, do both α and β derive strings beginning with a.
  2. At most one of α and β can derive ε
  3. If $\beta \Rightarrow^* \varepsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \Rightarrow^* \varepsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

All left-recursive or ambiguous grammars will produce an LL parsing table with at least one multiply-defined entry.

1. Rule 1 says we can't have ambiguous productions.

Same as saying FIRST(alpha) intersect FIRST(beta) = empty set
For condition 1 to be true, then we may have to left factor the grammar.

2. Same as 1, but for epsilon.
3. Same as saying if beta derives epsilon, then FIRST(alpha) and FOLLOW(A) are disjoint sets. This must be true because we add a production into M[A,a] for each a in FIRST(alpha) and we add a production to each M[A,b] for each b in FOLLOW(A) when beta derives epsilon.



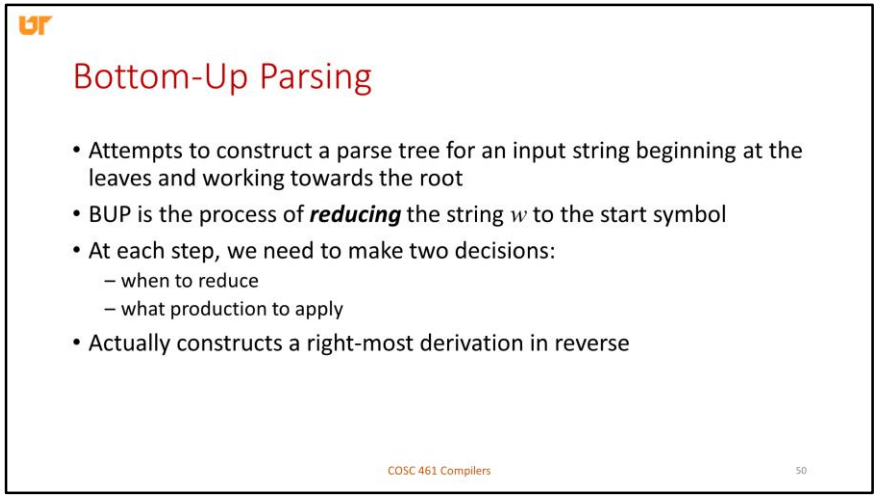
Consider the ambiguous if else statement we had seen before:

S -> iEtS | iEtSeS | a
E -> b

There is no left recursion in this grammar, but it needs to be left factored.

[Write up left factored grammar that is on the slide.]

[Do the FIRST and FOLLOW that is on the slide.]
[Construct the parsing table.]
[Show that the multiply-defined entry violates rule 3 on the previous slide.]



OK – that's it for top-down parsing. Now I want to start discussing bottom-up parsing.

BUP attempts to construct a parse tree for an input string beginning at the leaves and working towards the root. Actually, while it's convenient to describe parsing as the process of constructing a parse tree, a front end may carry out a translation without explicitly building a parse tree.

BUP is actually the process of reducing the string w to the start symbol of your grammar. By, definition, a reduction is the reverse step of a derivation. So – a derivation replaces the LHS of a grammar symbol with the RHS. A reduction replaces the RHS of a production with the LHS. So, what BUP parsing does is actually

At each step in the parse, we need to make two decisions:
When to reduce
What production to apply

This type of parsing produces a derivation in reverse. And actually, it's a rightmost derivation in reverse.

So, bottom-up parsers can be written for the set of grammars known as LR grammars. We have been looking at LL grammars before when we studied top-down parsing.

To be clear, all LL grammars can be parsed with a bottom-up parser.
All LR grammars can be parsed using a bottom-up parser, but some cannot be parsed with a top-down (LL) parser.

Different tools require different grammars. So, antlr is an example of a parser generator that generates an LL parser – so your grammar has to be in the right form. It creates a recursive descent parser, which is easier to debug by hand. YACC uses a bottom-up parser – so you can specify a broader range of grammars.

Example of Bottom-Up Parsing

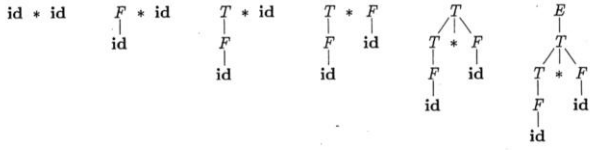


Figure 4.25: A bottom-up parse for **id * id**

Figure illustrates a sequence of reductions.

Write up the grammar:

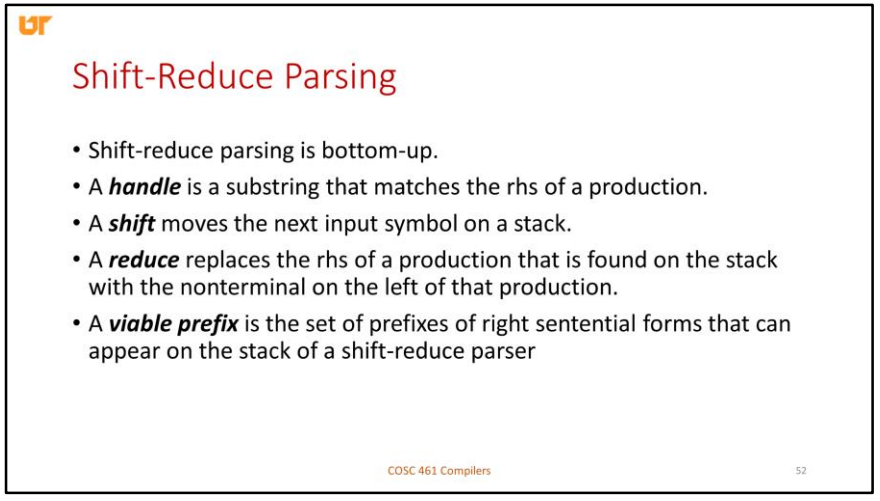
E -> E + T | T
T -> T * F | F
F -> (E) | id

First, we reduce id, by F->id.
Then, we reduce F, by T->F.

At step 3 we have a choice of reducing either T to E by the rule
E -> T, or reduce id by the rule F -> id. We choose the F->id.

Then, we reduce T->T*F, and the final step does E->T.

This parse tree actually corresponds to a rightmost derivation. So, if we write the derivation out, we have:
E => T => T * F => T * id => F * id => id * id



Now, I'd like to talk about a type of bottom-up parsing known as shift reduce parsing.
This is one of the most popular techniques for implementing BUP.
It uses a stack to hold grammar symbols (or states corresponding to parser configurations) and an input buffer to hold the rest of the string to be parsed.

Here is some vocabulary for shift reduce parsing:

A handle is a substring that matches the rhs of a production.
It always appears at the top of the stack.

The shift-reduce parser gets its name because it consists of 2 moves: a shift and a reduce.

A shift moves the next input symbol on the stack
A reduce replaces the rhs of a production that is found on the stack with the nonterminal on the left of that

production

And a viable prefix is the set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser.

Model of an LR Parser

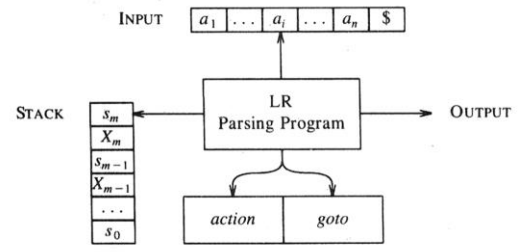


Fig. 4.35. Model of an LR parser.

Now, I want to talk about a type of shift-reduce parser called the LR parser. The L means the parser scans the input from left to right and the R means it produces a rightmost derivation.

Here is the model of a LR parser.

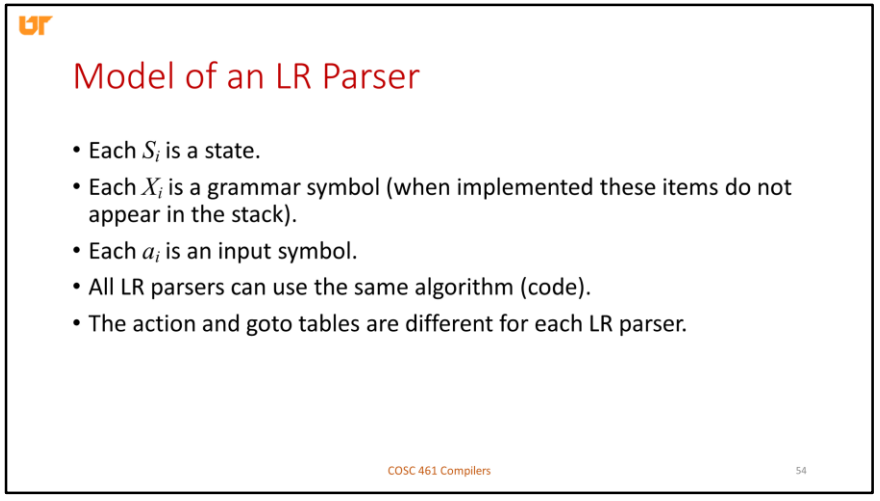
The model consists of an input, output, a stack, a driver program, and a parsing table.

Instead of grammar symbols (as in top down parsing), the stack holds states.

Each state corresponds to a sequence of grammar symbols. Each state summarizes the information of everything contained in the stack below it.

The action and goto blocks are parsing tables that the algorithm will use during the parse.

For each LR parser, we can use the same driver program, but the parsing tables changes from one parser to another.



Now, when we're talking about LR parsers we'll use these conventions:

Each S_i is a state.
Each X_i is a grammar symbol
And each a_i is an input symbol

Now, it's helpful to have a notation representing the complete state of the parser, i.e. its stack and remaining input. So, a configuration of an LR parser is a pair: ($s_0X_1s_1X_2s_2 \ldots X_ms_ma_ia_{i+1}a_n\$$) (where we have states and grammar symbols combined with the remaining input) Since each state is associated with a grammar symbol, we do not need to hold the grammar symbols on the stack.

The configuration then becomes: ($s_0s_1 \ldots s_m, a_ia_{i+1}a_n\$$) (series of states and the remaining inputs)

And this configuration represents the right-sentential form (i.e. sequence of terminals and non-terminals in a

rightmost derivation):
$X_1X_2 \ldots X_m a_i a_{i+1} a_n$

Another thing to note is that the input symbols in an LR parser are tokens, not characters.

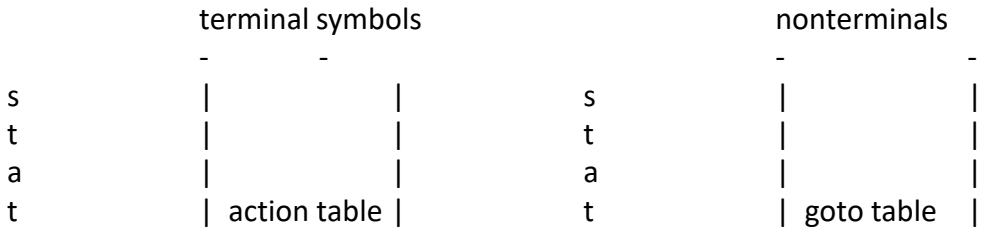
All LR parsers can use the same algorithm (i.e. source code).

A shift pushes a state on the stack and processes an input symbol.
A reduce pops some states off the stack and pushes one back on.

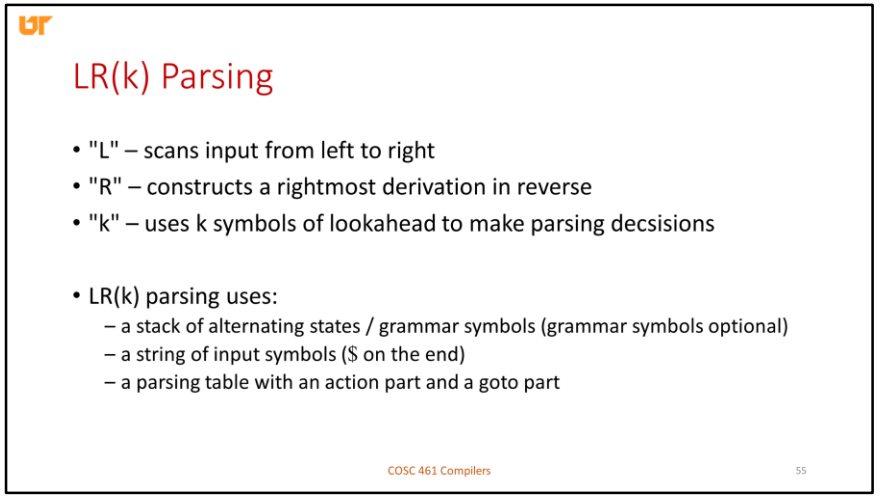
An LR parser uses two tables, an action table and a goto table.

The action table takes as arguments a state i and a terminal a, and it will index actions that tell the algorithm to either:
a) Shift states onto the stack
b) Reduce by popping the rhs of a production off the stack and replacing it with the non-terminal on the lhs
c) Accept the input string
d) Or produce an error

On the other hand, the goto table is indexed by a state and a non-terminal, will map states and non-terminals to another state.



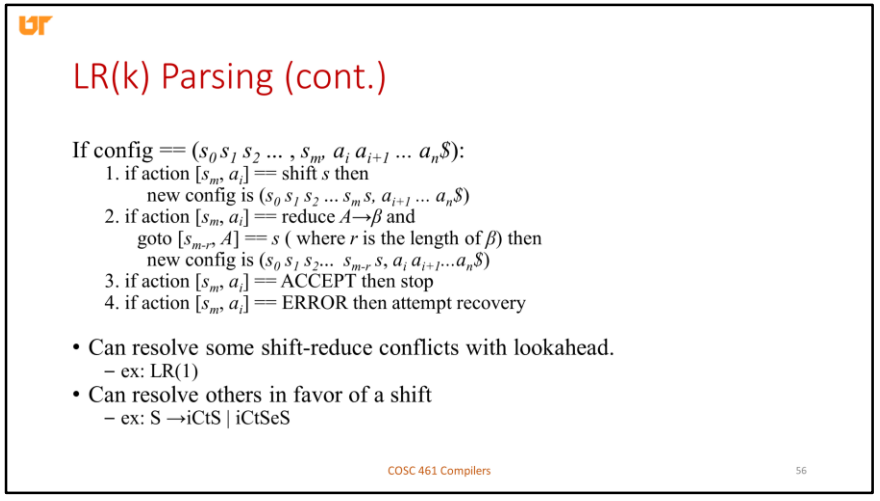
e s	| | | -	| | | -	e s	| | | -	| | | -



One type of shift-reduce parsing is an LR(k) parser.

This means we:
Scan input from left to right
Construct a rightmost derivation in reverse
And use k symbols of lookahead to make parsing decisions.

LR(k) parsing uses:
A stack of alternating states/grammar symbols (just as the model describes)
A string of input symbols with $ on the end. $ on the end of the input again represents the end of the input.
And a parsing table with an action part and a goto part.



Here is the algorithm for LR(k) parsing.

We consider a configuration ($s_0 s_1 \ldots s_m$, $a_i a_{i+1} \ldots a_n\$$) where si is a state, and a_i is your current input

If the action table indexed by s_m and a_i goes to a shift and state s, then the new config shifts state s onto the stack, and now a_{i+1} is the next input symbol.

If action [s_m , a_i] goes to reduce A -> beta and goto [s_{m-r}, A] is s (where r s the length of beta), then we pop r states off the stack, and push the state s (the state from the goto table) onto the stack. We don't consume any input.

If action [s_m , a_i] goes to accept, then we accept.
If action [s_m , a_i] goes to error, then we can attempt a recovery, which basically means we continue parsing to find

more syntax errors, but we do not provide a translation.

If there are conflicts with this algorithm, say we can choose whether or not to shift or reduce, there are a few ways to resolve this. Sometimes, we might be able to resolve the conflict by using a symbol of lookahead – this is what is known as an LR(1) parser.

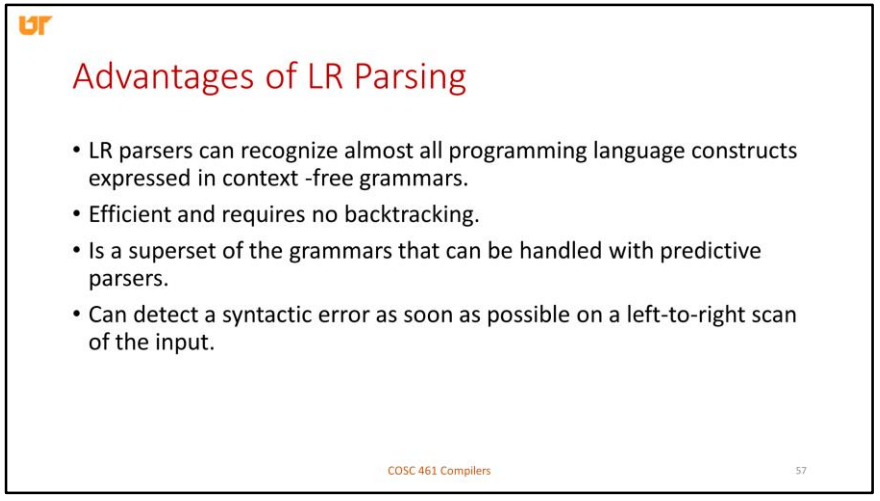
Often times, the default move is just to shift.
For example, in this grammar (which describes if then and if then else statements):

S -> iCtS | iCtSeS

Once the iCtS has been parsed and there is an e as the lookahead symbol:

(a) if shift then we choose iCtSeS rhs,
(b) if reduce then we choose ICtS rhs.

So, often the default is just to shift to match the longer production.



OK – so that's the algorithm for LR parsing. There are a number of advantages to using LR parsers:

1) LR parsers can recognize almost all programming language constructs expressed in context-free grammars. Constructs not recognized by an LR parser just don't seem to be used very often.

2) They are efficient and do not require any backtracking. For instance, the LR(1) parsers only need 1 symbol of lookahead.

3) LR parsers accept a superset of the grammars that can be handled with predictive parsers. The key here is that LR parsers collect the rhs of a production before making a parsing decision, so this allows a wider range of languages to be specified.

Consider the production: [A -> BCD]

A predictive parser has to make a parsing decision when only one terminal symbol is visible on the right hand side.
A bottom-up parser will make a parsing decision after seeing the entire right-hand side (rhs) and an additional terminal following the rhs.

4) LR parsers detect an error ASAP on a left-to-right scan of the input. If an input sequence does not match the RHS of one of the productions in your grammar – you'll know it immediately. Makes error handling easier, which we will discuss more later.

LR Parsing Example

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

[Put up slide for LR(k) Parsing on the screen.
Put up parse table via transparency of Figure 4.37.
Identify action (left) and goto (right) parts of the parse table.
Go through the derivation of id*id+id.
Put up derivation via transparency of Figure 4.38 to check.
Show it is a rightmost derivation in reverse.

Parsing Table for LR Parsing Example

STATE	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 4.37. Parsing table for expression grammar.

LR Parsing Example

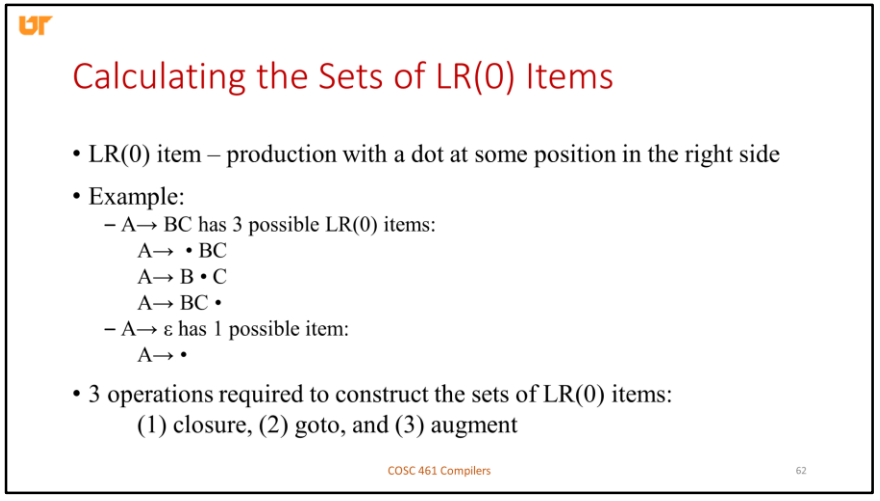
- It produces rightmost derivation in reverse:

$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow E + F \\
&\Rightarrow E + id \\
&\Rightarrow T + id \\
&\Rightarrow T * F + id \\
&\Rightarrow T * id + id \\
&\Rightarrow F * id + id \\
&\Rightarrow id * id + id
\end{aligned}
$$

It produces rightmost derivation in reverse.

	Stack	Symbols	Input	Action
(1)	0		id * id + id $	shift
(2)	0 5	id	* id + id $	reduce by F -> id
(3)	0 3	F	* id + id $	reduce by T -> F
(4)	0 2	T	* id + id $	shift
(5)	0 2 7	T *	id + id $	shift
(6)	0 2 7 5	T * id	+ id $	reduce by F -> id
(7)	0 2 7 10	T * F	+ id $	reduce by T -> T * F
(8)	0 2	T	+ id $	reduce by E -> T
(9)	0 1	E	+ id $	shift
(10)	0 1 6	E +	id $	shift
(11)	0 1 6 5	E + id	$	reduce by F -> id
(12)	0 1 6 3	E + F	$	reduce by T -> F
(13)	0 1 6 9	E + T	$	reduce by E -> E + T
(14)	0 1	E	$	accept

Shows stack and input contents during parse. Also shown for clarity are the sequences of grammar symbols corresponding to states held on the stack.



OK – now I want to talk about how we create the LR parsing table.

For this algorithm, we need to maintain states to keep track of where we are in a parse. So, we first have to construct the set of LR(0) items.

An LR(0) item is just a production with a dot at some position in the rhs of the production. This dot indicates how much of the production we have seen at a given point in the parsing process.

For example, in the production A->BC, there are 3 possible LR(0) items.
And A->E has 1 possible item, which is just A->dot

The items can be viewed as the set of states in the DFA representing the parser.

There are three operations we need to construct the sets of LR(0) items.

closure
goto
augment

Computation of CLOSURE

```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for ( each item A → α·Bβ in J )
            for ( each production B → γ of G )
                if ( B → ·γ is not in J )
                    add B → ·γ to J;
    until no more items are added to J on one round;
    return J;
}
```

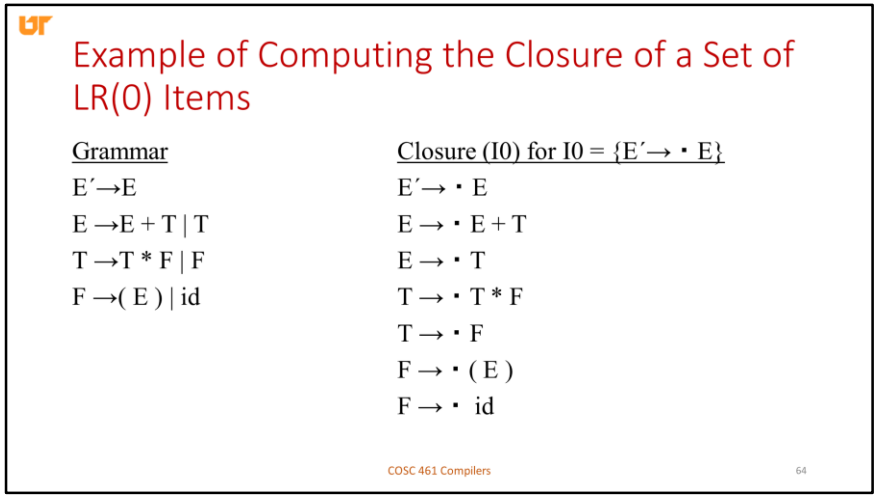
Figure 4.32: Computation of CLOSURE

Here is the definition of closure for a set of items I.

This is similar to the epsilon-closure function we saw earlier when constructing an NFA from a DFA.

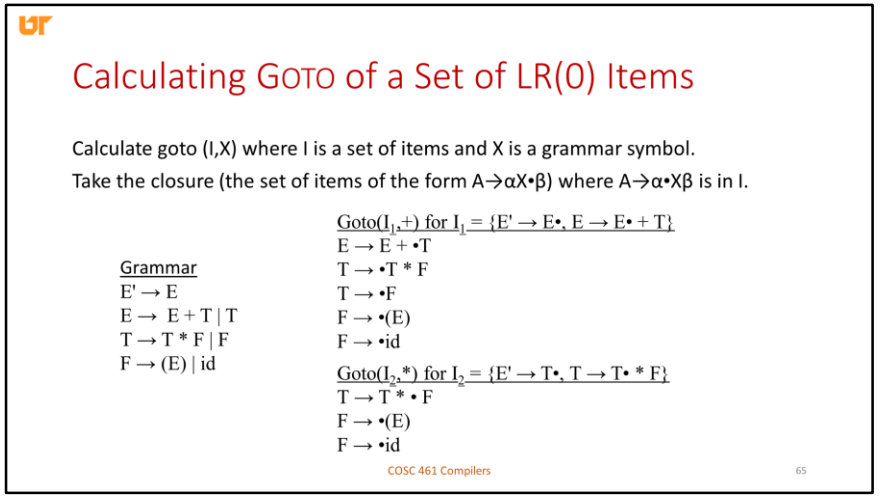
So, in this algorithm, J is the new set of items.

Initially we add every item in I into J. Then, for every item A-> alpha dot B beta, whenever there is a production B->gamma in G, we add B -> dot gamma to closure(I), if it's not already there. And we just keep applying this rule until no more items can be added.



So, given this grammar. If I is the set of one item { E' -> dot E }, then closure(I) contains the set of items shown here.

So, E' -> dot E is put into closure(I) immediately. So, since this has a dot immediately to the left of an E, we look for all the productions with E on the LHS. So, in the first iteration, we add E -> dot E + T and E -> dot T. Now, we continue with the new items. We don't have to add anything for E -> dot E + T, but we also have E -> dot T with the dot immediately to the left of the T. So, we add T -> dot T * F, and T -> dot F. And now, since we have a dot immediately to the left of the F in T -> dot F, we add F -> dot (E) and F -> dot id. After that, there are no more items to be added.



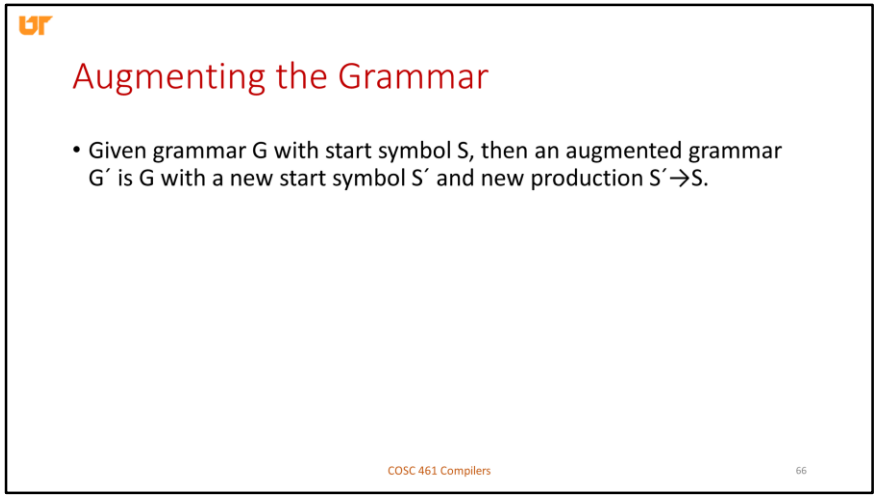
The goto operation takes a set of items and a grammar symbol as its arguments. Now, to calculate goto, we just move the dot in the items in I after the grammar symbol X, and take the closure.

So, for example, using the same grammar, if we want to calculate goto (I_1, +) for I_1 = { E' -> E dot, E -> E dot + T }
Just move the dot past the + in each item, and then take the closure of the resulting items.

First, note that, in the first item, E' -> E dot, there is no dot to the left of a plus, so we do not add any items to goto(I_1, +) for that item.
But we also have E-> E dot + T. For that one, we move the dot to right of the plus, and calculate the closure for the new item E->E + dot T. And taking the closure adds …

Similarly, for goto(I_2, *) for I_2 = { E' -> T dot, T -> T dot * F }, we do:
T->T * dot F, and taking the closure adds F -> dot (E) and F -> dot id

Intuitively, the goto operation is similar to taking a transition between states in the DFA, it's like the move operation.



Remember that in bottom-up parsing, we start with the string of terminals, and try to reduce the string back to the start symbol. In some cases, the start symbol in a grammar might appear on the rhs of some production. In that case, the parser wont know whether or not to accept the string at that point or try to reduce the start symbol.

To avoid this issue, we can do something known as augmenting the grammar. So, to augment a grammar G with start symbol S, we simply create a new start symbol S', and a new production S' -> S. In this way, S' will never appear on the rhs of a production, and the parser will always know to accept the string when it sees S', and reduce the string when it sees the old start symbol S.

Computation of Canonical Collection of Sets of LR(0) Items

```
void items(G') {
    C = CLOSURE({[S' → ·S]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C on a round;
}
```

Figure 4.33: Computation of the canonical collection of sets of LR(0) items

OK – here's the algorithm to compute the canonical collection of sets of LR(0) items. We can use this set of items to construct the action and goto tables for the LR(0) parsing algorithm.

Get the grammar from slide 58. First, augment the grammar:

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

The algorithm says, for each set of items I in C, add goto(I,X) to C, if its not already there.

So, starting with I_0 = {
E' → dot E
E → dot E + T
E → dot T
T → dot T * F
T → dot F
F → dot (E)
F → dot id
}

I_1 = goto(I_0, E) = {
E' -> E dot
E' -> E dot + T
}

I_2 = goto(I_0, T) = {
E -> T dot
T -> T dot * F
}

I_3 = goto(I_0, F) = {
T -> F dot
}

I_4 = goto(I_0, '(') = {
F -> (dot E)
E → dot E + T

E → dot T
T → dot T * F
T → dot F
F → dot (E)
F → dot id
}

I_5 = goto(I_0, id) = {
F -> id dot
}

I_6 = goto(I_1, +) = {
E -> E + dot T
T → dot T * F
T → dot F
F → dot (E)
F → dot id
}

I_7 = goto(I_2, *) = {
T -> T * dot F
F → dot (E)
F → dot id
}

I_8 = goto(I_4, E) = {
F -> (E dot)
E → E dot + T
}

goto(I_4, T) = I_2
goto(I_4, F) = I_3
goto(I_4, '(') = I_4
goto(I_4, id) = I_5

I_9 = goto(I_6, T) = {
E -> E + T dot
T -> T dot * F
}

goto(I_6, F) = I_3
goto(I_6, '(') = I_4
goto(I_6, id) = I_5

I_{10} = goto(I_7, F) = {
T -> T * F dot
}

goto(I_7, '(') = I_4
goto(I_7, id) = I_3

I_{11} = goto(I_8, ')') = {
F -> (E) dot
}

goto(I_8, +) = I_6
goto(I_9, *) = I_7

[Leave this slide on the screen.
Write the grammar from the LR Parsing Example slide on the board.
Augment the grammar as well.
Do the items for I0.
Construct the LR(0) items by calling on students after making them close their text.
]

LR(0) Automaton for Example Grammar

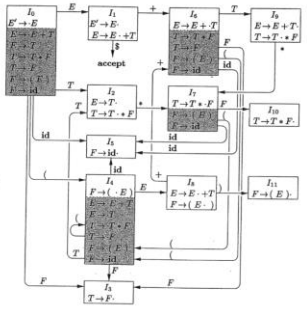


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

Put this up to check work.

Analogy of Calculating the Set of LR(0) Items with Converting an NFA to a DFA

- Constructing the set of items is similar to converting an NFA to a DFA
  - Each state in the NFA is an individual item
  - $\text{closure}(I)$ for a set of items is the same as $\varepsilon\text{-closure}$ for a set of NFA states
  - Each set of items is now a DFA state and $\text{goto}(I,X)$ gives the transition from I on symbol X

Read the slide

LR(0) Automaton for Example Grammar

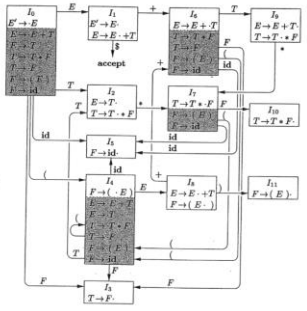
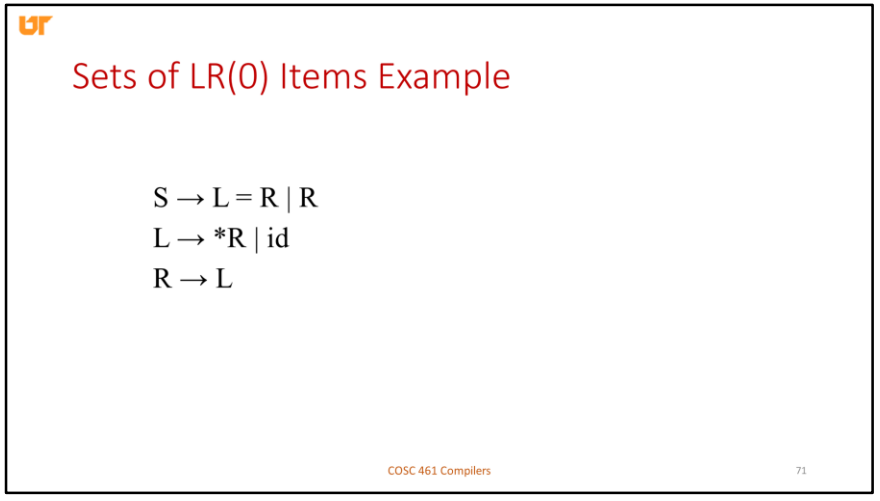


Figure 4.31: LR(0) automaton for the expression grammar (4.1)

Describe and compare to NFA to DFA



OK – let's try another example.

In this grammar, the L indicates an l-value (address) and the R indicates an r-value (value).
The * is the dereference operator

S -> L = R
S -> R
L -> *R
L -> id
R -> L

Augment the grammar:

S' -> S

S -> L = R | R
L -> *R | id
R -> L

I_0 = {
S' -> dot S
S -> dot L = R
S -> dot R
L -> dot *R
L -> dot id
R -> dot L
}

I_1 = goto(I_0, S) = {
S' -> S dot
}

I_2 = goto(I_0, L) = {
S -> L dot = R
R -> L dot
}

I_3 = goto(I_0, R) = {
S -> R dot
}

I_4 = goto(I_0, *) = {
L -> * dot R
R -> dot L

L -> dot * R
L -> dot id
}

I_5 = goto(I_0, id) = {
L -> id dot
}

I_6 = goto(I_2, =) = {
S -> L = dot R
R -> dot L
L -> dot * R
L -> dot id
}

I_7 = goto(I_4, R) = {
L -> * R dot
}

I_8 = goto(I_4, L) = {
R -> L dot
}

goto(I_4, *) = I_4
goto(I_4, id) = I_5

goto(I_6, R) = I_9 = {
S -> L = R dot
}

goto(I_6, L) = I_8
goto(I_6, *) = I_4
goto(I_6, id) = I_5

Canonical LR(0) Collection Example

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id} \cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$

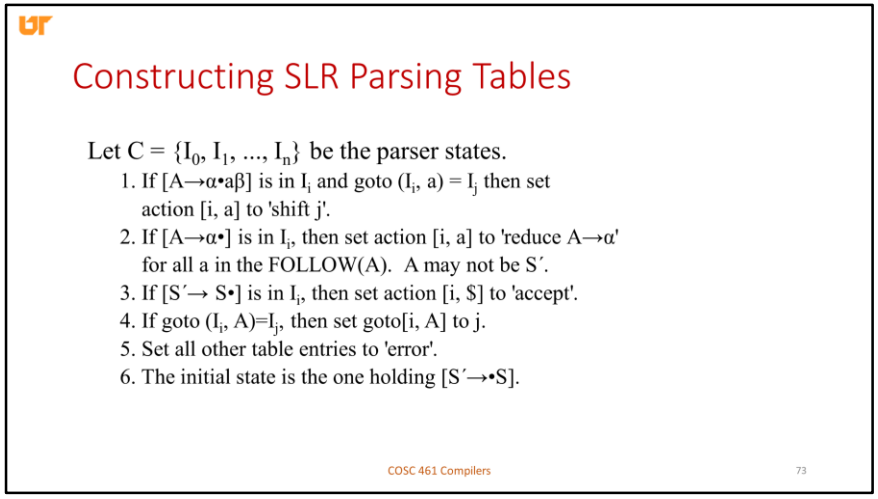
I_7: $L \to *R \cdot$

I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

Fig. 4.39. Canonical LR(0) collection for grammar (4.49).

Check that sets of items students did was correct.



So, as I had mentioned, we can use the canonical LR(0) items to construct the parsing tables for simple LR parsers.

Here is the algorithm for doing this once you've constructed the set of canonical LR(0) items, C for your input grammar. For each state I_i, we follow these steps.

So, basically, we put a shift in the table when we have a goto on a terminal. So, if I_i has a production A -> alpha dot a beta, and goto(I_i, a) = I_j, then in the table, we set action[i, a] to 'shift j'

We add reduces when the entire RHS of a production has been seen. Thus, for this rule, we have to calculate FOLLOW(A) because we want to add a reduce to the table when we're at a symbol that follows the RHS of a production.

Specifically, if we have a production A -> alpha dot in I_i, then we set action [i, a] to 'reduce A->alpha' for all the terminal symbols a in FOLLOW(A). Our non-terminal A on the LHS cannot be the start symbol.

If we have a production S' -> S dot, then we set action[i, $] to 'accept'

For the goto table, if we have an item where goto(I_i, A) = I_j where A is a non-terminal, then we set goto[i, A] to the state j.

Just like the LL(1) tables, all other entries in the tables are errors.

And we set the initial state to be the one holding S' -> dot S, which, in our examples, will always be I_0

Write the following productions on the board with numbers.
Note that each production has a single rhs.
1). E -> E + T
2). E -> T
3). T -> T * F
4). T -> F
5). F -> (E)
6). F -> id

Calculate the FIRST and FOLLOW of each nonterminal.

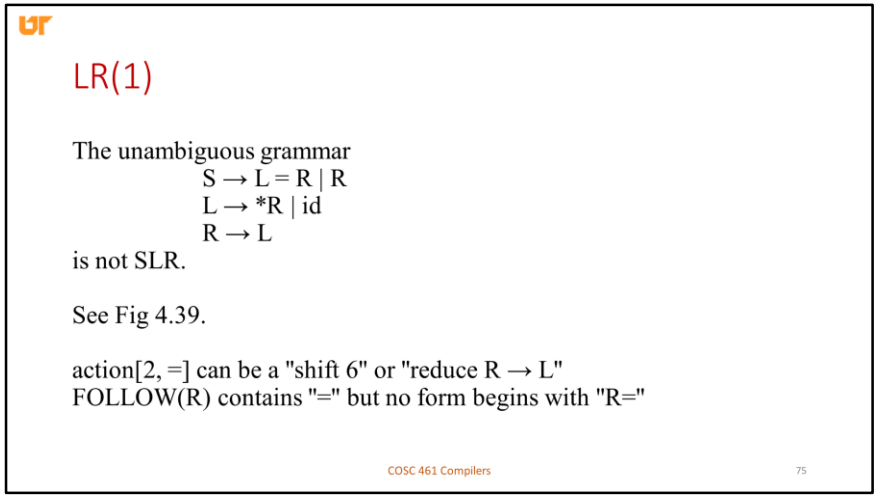
FIRST(E) = { (, id } FOLLOW(E) = { +,), $ }
FIRST(T) = { (, id } FOLLOW(T) = { *, +,), $ }
FIRST(F) = { (, id } FOLLOW(F) = { *, +,), $ }

[Build the SLR parsing table and check with Figure 4.37 transparency.]

Parsing Table for LR Parsing Example

STATE	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 4.37. Parsing table for expression grammar.



Up to now, we have been talking about SLR or simple LR parsers. These are a good starting point for talking about LR parsing, but there are more powerful LR parsers. For instance, there are some grammars for which we cannot construct an SLR parsing table, even if the grammar is unambiguous.

The example shown on the slide is an unambiguous grammar that is not SLR.

Calculate the FIRST and FOLLOW.

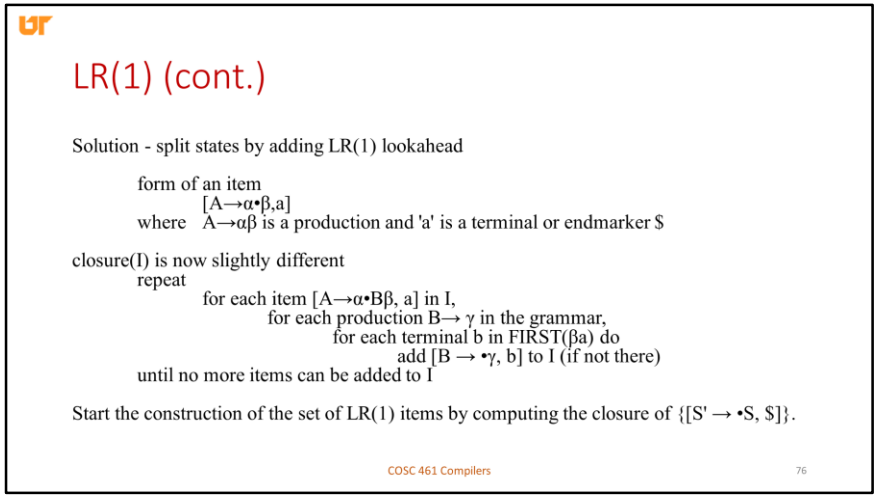
FIRST(S) = { *, id } FOLLOW(S) = { $ }
FIRST(L) = { *, id } FOLLOW(L) = { =, $ }
FIRST(R) = { *, id } FOLLOW(R) = { =, $ }

Look at I2.

Since I_2 has the production S -> L•=R, and since goto(I_2, =) = I_6, then action[2,=] is a shift to I_6.
However, we also have, R -> L• and FOLLOW(R) contains an '=', so action[2,=] should also be "reduce R -> L"

So, we have a shift/reduce conflict, despite the fact that the grammar is unambiguous.

The conflict occurs because the simple LR construction method is not powerful enough to accept the language specified by this grammar. Specifically, it cannot remember enough LHS context to decide what action the parser should take on input =, having seen a string reducible to L. Next, we'll talk about a couple different methods that can address these issues and succeed on a larger class of LR grammars.



The solution is to extend the previous LR techniques to split the states by adding a symbol of lookahead to each item. We call these items LR(1) items, or canonical LR items.

These LR(1) items look like this:

A -> alpha dot beta , a, where A -> alpha beta is a production and 'a' is a terminal or $. The 'a' terminal is the lookahead, which, in this context means something that might follow the RHS.

So, in the table construction, for productions of the form:

[A -> alpha • beta, a], where beta is not epsilon there is no effect because this would result in a shift or goto anyway.

But an item of the form

[A -> alpha •, a] calls for a reduction by A -> alpha only if the next input symbol is a. We do not reduce on all terminals that might follow A.

Now, for LR(1) construction, the closure of a set of items I is slightly different. We still add items when we have a dot immediately to the right of a non-terminal, but we add different items for each terminal b in FIRST(beta a). If beta is epsilon, then we just add [B -> dot gamma, a].

LR(1) Example

(0) 1. S' → S
(1) 2. S → CC
(2) 3. C → cC
(3) 4. C → d

I_0:	[S' → •S, $]	goto (S)	= I_1
	[S → • CC, $]	goto (C)	= I_2
	[C → • cC, c/d]	goto (c)	= I_3
	[C → • d, c/d]	goto (d)	= I_4
I_1:	[S' → S •, $]		
I_2:	[S → C • C, $]	goto (C)	= I_5
	[C → • cC, $]	goto (c)	= I_6
	[C → • d, $]	goto (d)	= I_7

OK – let's try this with an example.

Here's our grammar:

S -> CC
C -> cC | d

Calculate FIRST(S) = {c, d} and FIRST(C) = {c, d}.

For each closure, write up what A, alpha, B, beta, a are.

First example:
S' -> dot S, $

A = S', alpha = E, B = S, beta = E, a = $

We have the production S -> CC, and we need to add that for each terminal in FIRST(beta a). Since beta is E, FIRST(beta a) can only be a, which is $

Next:
A = S, alpha = E, B = C, beta = C, a = $

So, then we add C -> dot cC for each terminal in FIRST(C$) = FIRST(C) = {c,d}
Similarly, we add C -> dot d for both c and d as the lookahead symbol.

I_0 = {
S' -> dot S, $
S -> dot CC, $
C -> dot cC, c/d
C -> dot d, c/d
]

The rest of the construction proceeds as the LR(0) items went. We do goto(I_0, S) = I_1

goto(I_0, S) = I_1 = {
S' -> S dot, $
}

And goto(I_0, C) = I_2 = {
S -> C dot C, $,

A = S, alpha = C, B = C, beta = E, a = $

goto(I_0, C) = I_2 = {
S -> C dot C, $,
C -> dot cC, $,
C -> dot d, $
}

goto(I_0, c) = I_3 = {
C -> c dot C, c/d,
C -> dot cC, c/d,
C -> dot d, c/d
}

goto(I_0, d) = I_4 = {
C -> d dot, c/d
}

goto(I_2, C) = I_5 = {
S -> CC dot, $
}

Now, we have goto(I_2, c), and we have C -> c dot C, which we had seen previously, but we have a different lookahead symbol, so we have to create a new set we'll call I_6:

A = C, alpha = c, B = C, beta = E, a = $

goto(I_2, c) = I_6 = {
C -> c dot C, $,
C -> dot cC, $,
C -> dot d, $

}

goto(I_2, d) = I_7 = {
C -> d dot, $
}

goto(I_3, C) = I_8 = {
C -> cC dot, c/d
}

goto(I_3, c) = I_3
goto(I_3, d) = I_4

goto(I_6, C) = I_9 = {
C -> cC dot, $
}

goto(I_6, c) = I_6
goto(I_6, d) = I_7

LR(1) Example (cont.)

I_3:	[C → c•C, c/d]	goto (C)	= I_8
	[C → •cC, c/d]	goto (c)	= I_3
	[C → •d, c/d]	goto (d)	= I_4
I_4:	[C → d•, c/d]		
I_5:	[S → CC•, $]		
I_6:	[C → c•C, $]	goto (C)	= I_9
	[C → •cC, $]	goto (c)	= I_6
	[C → •d, $]	goto (d)	= I_7
I_7:	[C → d•, $]		
I_8:	[C → cC•, c/d]		
I_9:	[C → cC•, $]		

The GOTO Graph for Grammar

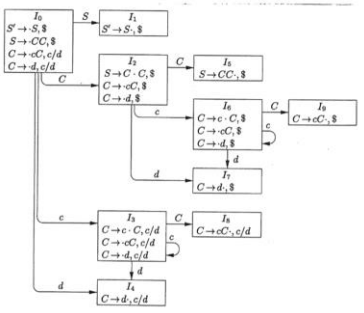


Figure 4.41: The GOTO graph for grammar (4.55)

So this graph shows how transitions can be made from one state to another.

Constructing the LR(1) Parsing Table

Let C = $\{I_0, I_1, ..., I_n\}$

1. If $[A \rightarrow \alpha \bullet a\beta]$ is in Ii and goto(I_i, a) = I_j then set action[i, a] to "shift j".
2. If $[A \rightarrow \alpha \bullet, a]$ is in I_i, then set action[i, a] to 'reduce $A \rightarrow \alpha$'.
   (A may not be S')
3. If $[S' \rightarrow S \bullet, \$]$ is in I_i, then set action[i, $] to "accept."
4. If goto(I_i, A) = I_j, then set goto[i, A] to j.
5. Set all other table entries to error.
6. The initial state is the one holding $[S' \rightarrow \bullet S, \$]$

Here is the construction for an LR(1) parsing table.

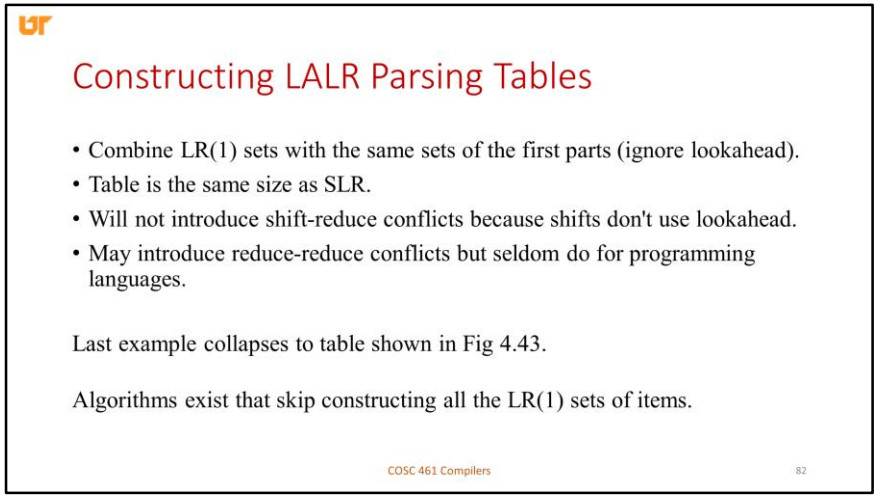
The only real difference from what we had seen before is step 2. For SLR construction, we added reduce A->alpha for every 'a' in FOLLOW(A). Now, we just add the reduce action for the particular symbol of lookahead specified for this item.

[Produce table from set of items on previous slide.]

Canonical Parsing Table

STATE	*action*			*goto*	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Fig. 4.42. Canonical parsing table for grammar (4.55).



There is one more type of LR parser I want to talk about – called lookahead or LALR parsing.

The language generated by the above grammar can be represented as the RE c*dc*d

Consider states 4 and 7. State 4 reduce by the rule C -> d on lookaheads c/d. State 7 has the same reduction on lookahead $.

This is because state 4 will be reached after reading the first set of c's followed by the d, i.e., c*d.

So, the next input should be either a c/d. $ would be invalid. The parser enters state 7 on reading the string c*dc*d, so any lookahead other than $ is invalid.

A "lookahead" LR parser just combines LR(1) items with the same sets of first parts. So, basically, it ignores all the lookahead symbols. The resulting parser will be similar to the simple LR parser we saw earlier, and will have the same

size table as the SLR parser.

However, it is not exactly the same as SLR, and is in fact more powerful than SLR, because the union of lookaheads may be smaller than the FOLLOW(A), where A -> alpha.

But it is less powerful than the LR(1) parser. If we combine items 4 and 7 from the LR(1) parser, we create a state called I_{47} that reduces on any lookahead symbol c, d, or $. This is not always correct, but the error will always eventually be caught.

Another thing to note is that the LALR construction cannot introduce any shift-reduce conflicts that were not already present in the LR(1) parse table. That is because the combined states in the LALR parser table share the same core items that the LR(1) parse table uses (i.e. they are the same, except for the lookahead symbols) and shifts do not depend on the lookahead symbols.

But, it can result in a reduce/reduce conflict not present in LR(1).

Combine the sets of items with common cores on the board.
Gotos also do not depend on lookaheads – so there's no chance of a conflict in the goto table.

For this construction, we can combine:

States 3 and 6
States 4 and 7
States 8 and 9

Constructing LALR Parsing Tables

STATE	action			goto	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Fig. 4.43. LALR parsing table for grammar (4.54).

You can skip the section on "Efficient Construction of LALR Parsing Tables."

Using Ambiguous Grammars

$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow (E)$$
$$E \rightarrow id$$

instead of

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

- See figure 4.48
- Advantages:
  - Grammar is easier to read
  - Parser is more efficient

Now – I want to talk about how we can use ambiguous grammars during parsing.

Previously, we had established precedence and associativity by using more nonterminals and productions.

But there are some advantages to using ambiguous grammars:

(1) grammar is easier to read,
(2) because there are less productions, the parser is more efficient (fewer reductions, no E -> T or T -> F), and
(3) If we change precedence or associativity, there is no need to change the grammar.

Sets of LR(0) Items for an Augmented Expression Grammar

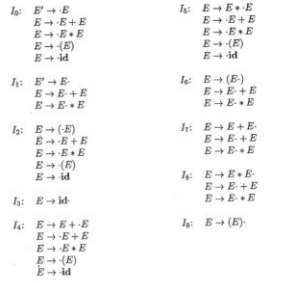


Figure 4.48: Sets of LR(0) items for an augmented expression grammar

Show this figure to check work and fill in the entries.

Go ahead and construct the parsing table for these items.

Notice the shift reduce conflict for states 7 and 8. These conflicts are generated for SLR and LR(1) and LALR parsing methods.

Using Ambiguous Grammars (cont.)

- Can use precedence and associativity to solve the problem
- See Figure 4.49

shift / reduce conflict in state action [7,+] = (s4, r1)
s4 = shift 4 or $E \rightarrow E\bullet + E$
r1 = reduce 1 or $E \rightarrow E + E\bullet$

id + id + id
↑ cursor here

action[7,*] = (s5,r1)
action[8,+] = (s4,r2)
action[8,*] = (s5,r2)

[Call on the students to fill in the entries, while keeping the previous slide (Figure 4.48) on the screen.]

FIRST(E)={(,id} FOLLOW(E)={+,*,),$}

Consider the input id + id * id

On this input, we reach the state:

Prefix:	Stack	Input
E + E	0 1 4 7	* id $

In this case, there is a shift-reduce conflict on action[7, *]. Because, however, we know that * has higher precedence

than +, we will go ahead and shift the * onto the stack so that we can reduce the * before the + is reduced.

What if the input is id + id + id? Since + is left associative, we want to reduce the + on the left (the one that we have already parsed in the example above) before shifting the next plus onto the stack.

Correct choices:

	+	*
7	r1	s5
8	r2	r2

This example shows how to use ambiguous grammars for SLR parse tables, but ambiguous grammars can be used in a similar way for the canonical LR and LALR.

Parsing Table for Ambiguous Grammar

STATE	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Figure 4.49: Parsing table for grammar (4.3)

Another Ambiguous Grammar

0. $S' \rightarrow S$
1. $S \rightarrow iSeS$
2. $S \rightarrow iS$
3. $S \rightarrow a$

See Figure 4.50

action[4,e] = (s5,r2)

Leave this as an exercise for the students.

Have students produce LR(0) set of items.

LR(0) States for Augmented Grammar

I_0: $S' \to \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_1: $S' \to S \cdot$

I_2: $S \to i \cdot SeS$
$S \to i \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_3: $S \to a \cdot$

I_4: $S \to iS \cdot eS$
$S \to iS \cdot$

I_5: $S \to iSe \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_6: $S \to iSeS \cdot$

Fig. 4.50. LR(0) states for augmented grammar (4.67).

Show this to check work.

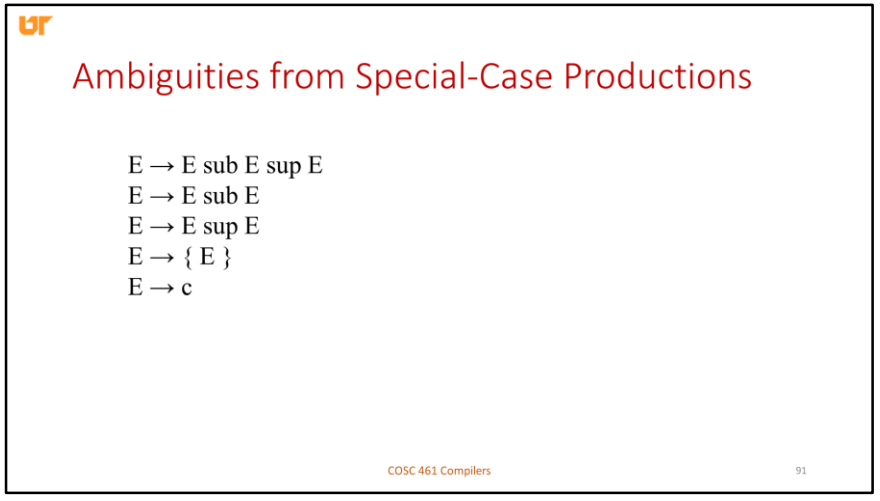
Have students produce SLR parsing table for LR(0) set of items in Figure 4.50.
Have students resolve conflict.

LR Parsing Table for "Dangling-Else" Grammar

STATE	action				goto
	i	*e*	*a*	$	*S*
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Fig. 4.51. LR parsing table for abstract "dangling-else" grammar.

Show this table to check work and that shift-reduce conflict was resolved correctly.



Here is a grammar that describes a language with subscript and superscripts on variables.

So: sub = subscript. sup = superscript.

Now, we want to be able to write statements like this:

a sub i sup 2 – and that would give us: {a^2}_{i}

But, with just the bottom four productions, this would never work. We would always get: a_{i^2}

So, we added a special case production:
E → E sub E sup E

That will allow us to have a superscript on an item with a subscript.

Ambiguities from Special-Case Productions

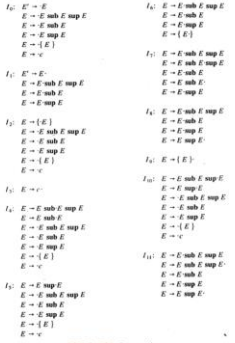
E → E sub E sup E
E → E sub E
E → E sup E
E → { E }
E → c

FIRST(E) = { '{', c }
FOLLOW(E) = { sub, sup, '}', $}

sub, sup have equal precedence
and are right associative

[Construct table from Figure B.
Show how conflicts are resolved.
Show Figure C to check the work.]
All shift/reduce conflicts are resolved as shifts in this case since the operators are right associative and have equal precedence.
The reduce/reduce conflicts are resolved in favor of the special case production.
[Show it works with c sub c sup c sub c \$.]

LR(0) Sets of Items for Example Grammar



Ambiguities from Special-Case Productions

E → E sub E sup E
E → E sub E
E → E sup E
E → { E }
E → c

FIRST(E) = { '{', c }
FOLLOW(E) = { sub, sup, '}', $}

sub, sup have equal precedence
and are right associative

action[7,sub] = (s4,r2)
action[8,sub] = (s4,r3)
action[11,sub] = (s5,r1,r3)
action[11,}] = (r1,r3)

action[7,sup] = (s10,r2)
action[8,sup] = (s5,r3)
action[11,sup] = (s5,r1,r3)
action[11,$] = (r1,r3)

[Construct table from Figure B.
Show how conflicts are resolved.
Show Figure C to check the work.]

This results in a table with many shift/reduce conflicts and some reduce / reduce conflicts.

The shift/reduce conflicts can be resolved as shifts since all the operators are right associative and have equal precedence.

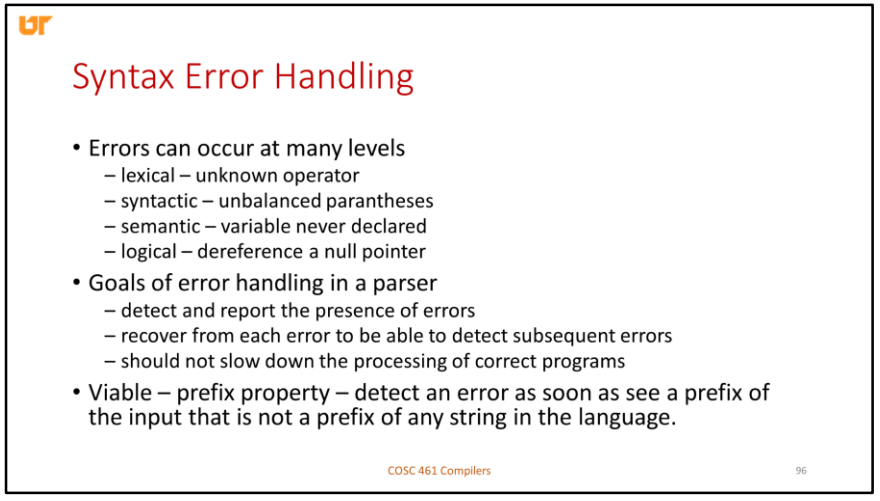
The reduce / reduce conflicts are always resolved in favor of the special case production we added – so that we match the special case if we see it.

[Show it works with c sub c sup c sub c \$.]

Parsing Table for Example Grammar

STATE	action						goto
	sub	**sup**	{	}	*c*	$	*E*
0			s2		s3		1
1	s4	s5				acc	
2			s2		s3		6
3	r5	r5		r5		r5	
4			s2		s3		7
5			s2		s3		8
6	s4	s5		s9			
7	s4	s10		r2		r2	
8	s4	s5		r3		r3	
9	r4	r4		r4		r4	
10			s2		s3		11
11	s4	s5		r1		r1	

Fig. C. Parsing table for grammar (4.25).



Before concluding the chapter, I'd like to cover some topics in error handling.

Programming errors can occur at many different levels. In general, in compilers we talk about four types of errors:

Lexical errors – e.g. we might have an unknown operator that the scanner does not know how to handle
Syntactic errors – these are errors detected by the parser. For instance, we might have unbalanced parantheses.
Semantic errors – these are errors detected by semantic analysis, including type checking. So, if a variable was not declared, or was used in a certain way that did not match it's type, this would count as a semantic error.
Logical errors – are errors that occur at runtime. For instance, dereferencing a null pointer.

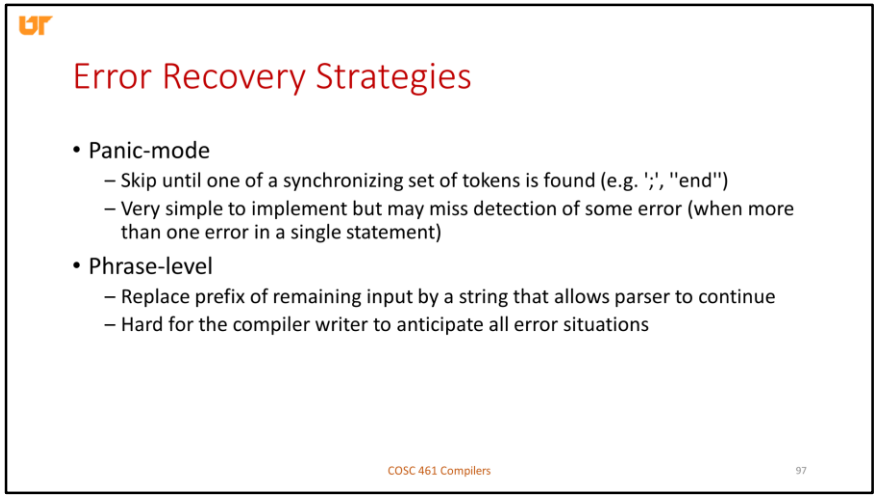
During compilation, most errors occur at the syntactic level. Some occur at the semantic level. Very few occur at the lexical level.

The main goals of error handling are to:

1) Detect and report the presence of errors in your program,
2) Recover from each error in a way that we can continue the compilation process, and perhaps detect additional errors
3) And, ideally, error checking should be fast. We should not spend too much time checking for errors when the input is actually a correct program.

One valuable property that is used in error recovery is the viable prefix property. This says we should detect an error as soon as we see a prefix of the input that is not a prefix of any string in the language. This helps to place error messages close to the point of the actual syntax error.

Often times, compilers will print the line with the syntax error with a pointer to the position (token) where the error was detected along with a message.

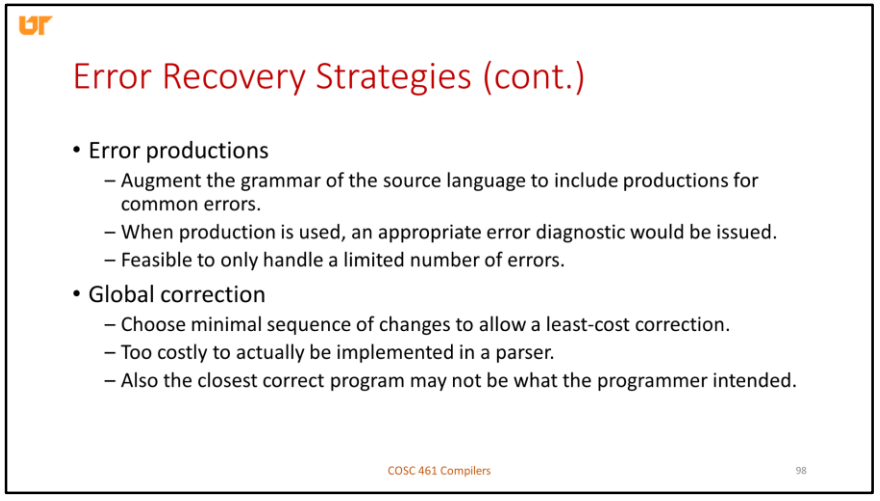


Now, let's talk about a few different error recovery strategies.
There are a few different types of error recovery strategies:

1) panic-mode recovery
This type keeps skipping over input symbols until one of a set of synchronizing tokens is found. For instance, it might skip until the ; or } is found. It's simple to implement, but it could miss some errors if there is more than one error before you reach one of the synchronizing tokens. It's also guaranteed to not go into an infinite loop.

2) phrase-level recovery
In this mode, we replace the prefix of the remaining input with some string that will allow the parser to continue looking for errors. This can detect more errors, but it could also possibly go into an infinite loop if we insert something that prevents the parser from eventually consuming an input symbol. It's also a little harder to implement because it's hard for the compiler writer to anticipate all error situations.

A bad job of error recovery may produce a lot of extra error messages. For instance, if one forgets a '}' in C, then when we start a new function, we will often get an error message for every declaration.

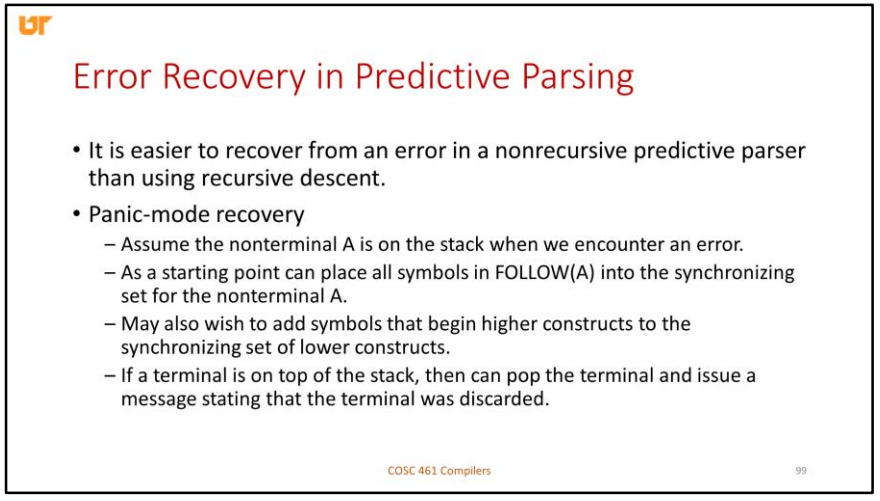


3) error productions
Another approach is to augment the grammar for our source language to include productions for common errors. When the error production is used, an appropriate error diagnostic can be issued.
This approach is feasible for handling only a limited number of errors because, not only do you have to come up with the error productions, but it will also increase the number of states in your parsing table.

For instance: var = expr instead of var := expr in Pascal.
Or handling if (var = 0) in C

4) global correction
This approach is basically error repair. It attempts to chose a minimal sequence of changes to allow for a least-cost correction. Global correction can require a lot of computation and is too costly to actually be implemented in the parser. Also, the closest correct program might not be what the programmer intended.

One example is the PL/I compiler developed by IBM. This compiler tried to guess what the programmer meant and attempt to fix every bug in the program and would spit back an executable. It often guessed wrong and people would have to go back and fix their programs anyway. So, it's often considered bad policy to try to do error repair in compilers.

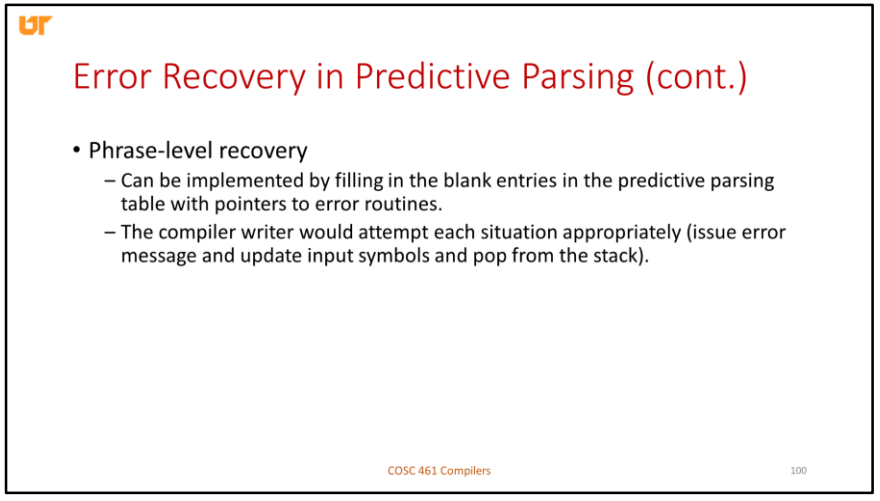


Easier to recover from error in non-recursive predictive parsing because the state is maintained in an explicit stack.

[show slide on Nonrecursive Predictive Parsing Example first to remind them.]
[show slide on Nonrecursive Predictive Parsing when describing errors. (slide 40)]

An error can occur when (a) a terminal symbol is on top of the stack and it does not match the next input symbol or (b) a nonterminal A is on top of the stack, a is the next input symbol, and M[A, a] is empty.

We can use panic mode recovery by skipping symbols until one in a set appears associated with A (A$\rightarrow$$\alpha$).
If we find such a terminal on the stack, then we pop it and issue an error message.



For phrase-level recovery we have to be careful when altering symbols on the stack or pushing symbols on the stack since this may lead to an infinite loop.

Synchronizing Tokens Added to Parsing Table

NONTERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id	synch	synch	$F \to (E)$	synch	synch

Fig. 4.22. Synchronizing tokens added to parsing table of Fig. 4.17.

This is used for panic-level recovery in predictive parsing.
The synchronizing tokens corresponding to the tokens in the follow set of the respective non-terminal are inserted into the table. Input symbols are skipped until a synchronizing token is reached.

So, here's how you use this table:

If the entry is blank, you generate an error and skip the input symbol to try to continue parsing. Otherwise, if it's synch, then you consider that you parsed the non-terminal on the stack, and pop it off.

Parsing and Error Recovery Moves Made by a Predictive Parser

STACK	INPUT	REMARK
$E	) id * + id $	error, skip)
$E	id * + id $	id is in FIRST(E)
$E'T	id * + id $	
$E'T'F	id * + id $	
$E'T'id	id * + id $	
$E'T'	* + id $	
$E'T'F*	* + id $	
$E'T'F	+ id $	error, M[F, +] = synch
$E'T'	+ id $	F has been popped
$E'	+ id $	
$E'T+	+ id $	
$E'T	id $	
$E'T'F	id $	
$E'T'id	id $	
$E'T'	$	
$E'	$	
$	$	

Fig. 4.23. Parsing and error recovery moves made by predictive parser.

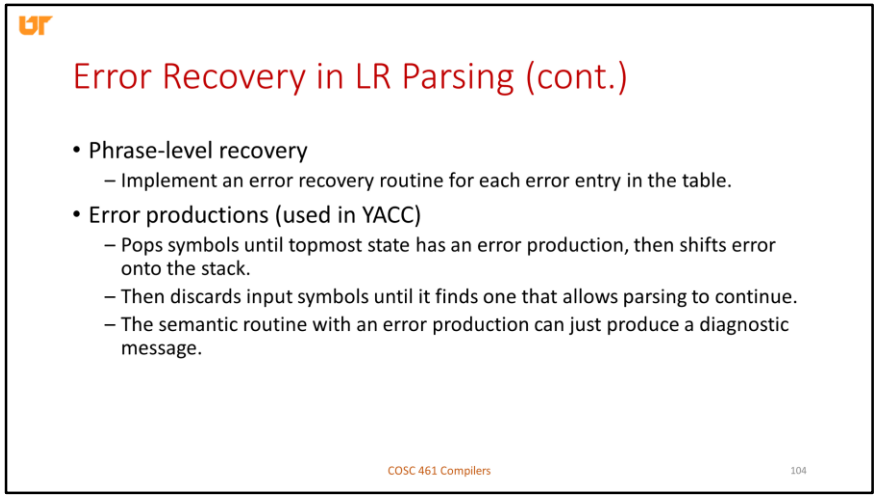
$')'$ is in the synchronizing set for E, but we cannot pop the first symbol.
Also, id is in the first of E, so we skip ')'.
$+$ is in the synchronizing set for F, so F is popped.

Error Recovery in LR Parsing

- Canonical LR Parser
  - Will never make a single reduction before recognizing an error.
- SLR and LALR Parsers
  - May make extra reductions but will never shift erroneous input symbol on the stack.
- Panic-mode Recovery
  - Scan down stack until a state with a goto on a particular nonterminal representing a major program construct (e.g. expression, statement, block, etc.) is found.
  - Input symbols discarded until one is found that is in the FOLLOW of the nonterminal.
  - The parser then pushes on the state in goto.
  - Thus, it attempts to isolate the phrase containing the error.

[show LR(k) Parsing slide first].
An LR parser detects an error when it finds an error (blank) entry
in the action parsing table.
There is never an error with the goto table since each production should
have a goto entry filled in for it.

Note that scanning down the stack for panic mode recovery means popping states.



For error productions in Yacc, it would pop states until it finds

A $\rightarrow$ $\bullet$ error α
Suppose α == ;.
Then it will discard symbols until a ';' is found and then it will discard
the ';' as well.
At that point it will do the reduction and continue the parse.

Yacc errors:

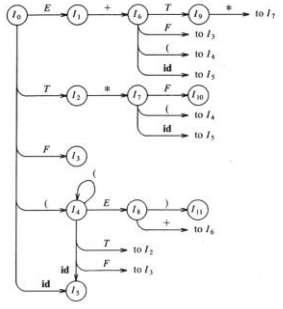
Error productions actually specified by the user
A -> dot error alpha

After an error, pops stack symbols until the topmost state has an error production. Then shift the error production on

the stack like you saw the error as a token in your input stream. This will execute the semantic action associated with the error.

Backup

Transition Diagram for Viable Prefixes



A Transition diagram of DFA D for viable prefixes.

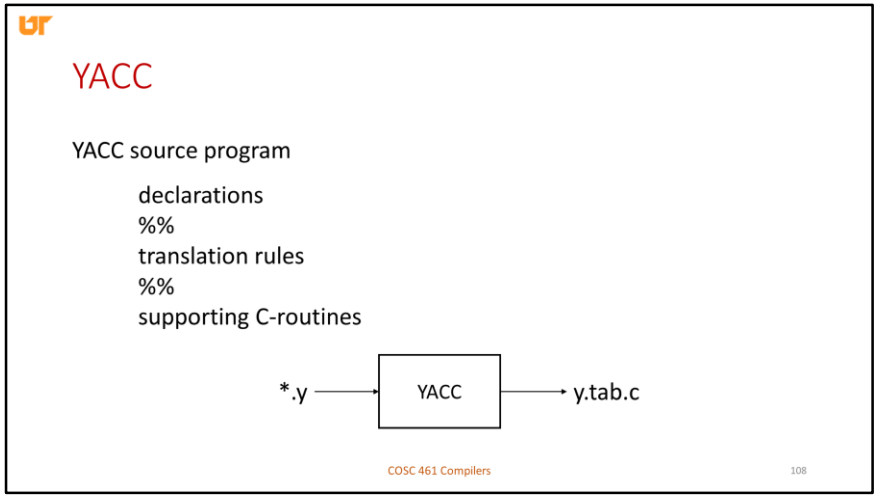
This is the state transition diagram for finding a viable prefix shifted on the stack of an SLR parser.

Nodes with no transitions only do reductions.

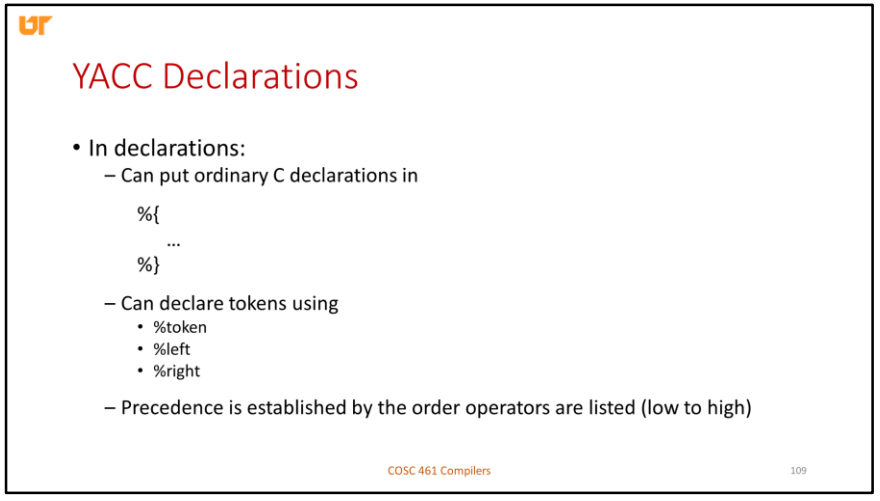
A viable prefix is a prefix of a right sentential form that can appear on the stack of a shift reduce parser.

Compaction of LR Parsing Tables

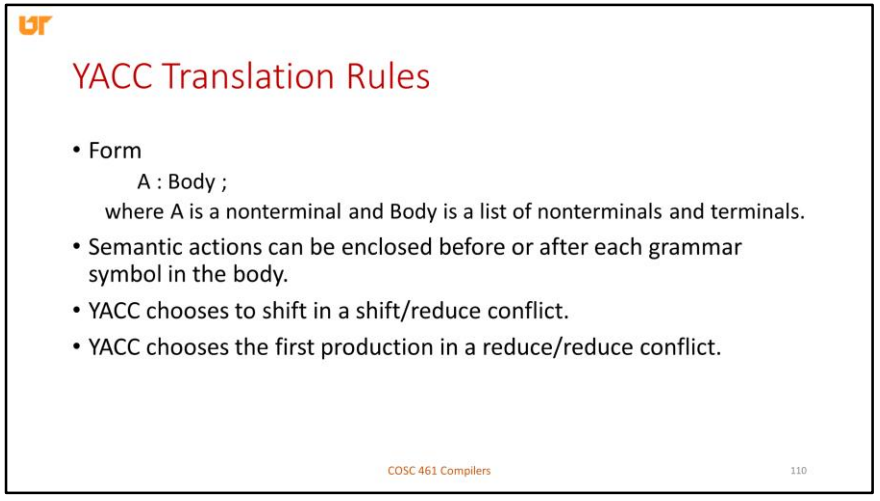
- A typical programming language may have 50 to 100 terminals and over 100 productions. This can result in several hundred states and a very large action table.
- One technique to save space is to recognize that many rows of the action table are identical. Can create a pointer for each state with the same actions point to the same location.
- Could save further space by creating a list for the actions of each state, where the list consists of terminal-symbol/action pairs. This would eliminate the blank or error entries in the action table. While this technique would save a lot of space, the parser would be much slower.



Can see that the syntax is somewhat similar to lex in that the sections are separated by \%\% and the first part is for declarations, the second is for rules, and the third is for supporting routines.
Yacc stands for Yet Another Compiler Compiler.
It uses the LALR parsing technique.
Yacc produces a routine called yyparse() in the y.tab.c file.
yyparse returns 0 if a string of tokens is in the language.
Otherwise it returns a nonzero value.
If one compiles with "gcc y.tab.c -ly", then no main() function is required since it is in the liby.a library.



\%token just says to make a symbol a token.
Charachter sequences specified in single or double quotes within the specification rules are taken to be tokens, and do not need to be explicitly defined using the \%token keyword.
All other character sequences are nonterminals.
\%left and \%right says make the symbol a token, but also indicate the associativity.
If ambiguous, then Yacc will resolve using rules specified on next slide.



Yacc chooses the first reduce in a reduce/reduce conflict.
So one could order productions to get the desired reduction.

The ':' separates the nonterminal on the lhs and the body which is the rhs.
The ';' ends a rule.
So a rule may span multiple lines.

The supporting C routines are called from actions in the translation rules.

Whenever yacc needs a token it calls yylex().
So yacc and lex are designed to work together since yylex() is the name
of the function produced in the file lex.yy.c by lex.

If we need a value (lexeme) associated with the token (not just the type),

then the function yylex() has to update the global variable yylval.
This allows nonterminals to have attributes, which will be described more in Chapter 5.

Typically yacc is used as a translator, not just as a recognizer.

YACC Translation Rules (cont.)

- When there is more than one rule with the same left hand side, a '|' can be used.

```
A : B C D ;
A : E F ;
A : G ;
```

⇒

```
A : B C D
  | E F
  | G
  ;
```


The | is just a short hand notation so the nonterminal on the left hand side does not have to be repeated.

Example of a YACC Specification

```
%token IF ELSE NAME                     /* defines multicharacter tokens */
%right '='                              /* low precedence, a=b=c shifts */
%left '+' '-'                           /* mid precedence, a-b-c reduces */
%left '*' '/'                           /* high precedence, a/b/c reduces */
%%
stmt        : expr ';'
            | IF '(' expr ')' stmt
            | IF '(' expr ')' stmt ELSE stmt
            ;                           /* prefers shift to reduce in shift/reduce conflict */
expr : NAME '=' expr                    /* assignment */
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '-' expr %prec '*'                    /* can override precedence */
| NAME
;
%% /* definitions of yylex, etc. can follow */
```


So '=' is right associative and '+', '-', '*', and '/' are left associative.
'*' and '/' have higher precedence since they are listed last.
Yacc will recognize the if-then-else production before the if-then since it resolves shift/reduce conflicts with a shift.
We were able to denote that the unary minus operation has the same precedence as the '*' operator.

The precedences and associativities are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:
1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of

the rule. If the \%prec construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict, and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

YACC Actions

- Actions are C code segments enclosed in { } and may be placed before or after any grammar symbol in the right hand side of a rule.
- To return a value associated with a rule, the action can set $$.
- To access a value associated with a grammar symbol on the right hand side, use $i, where i is the position of that grammar symbol.
- The default action for a rule is { $$ = $1; }

Yacc actions are like the C code fragments in Lex.
They can be used to make semantic checks, perform a translation, return a value, or take some other type of actions.

YACC Specification of a Desk Calculator

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line    :   expr '\n'         { printf("%d\n", $1); }
        ;
expr    :   expr '+' term     { $$ = $1 + $3; }
        |   term
        ;
term    :   term '*' factor   { $$ = $1 * $3; }
        |   factor
        ;
factor  :   '(' expr ')'      { $$ = $2; }
        |   DIGIT
        ;

%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

Fig. 4.59. Yacc specification of a simple desk calculator.

The grammar in this example has been defined to be unambiguous.
So we no longer need the precedence specifications.
So this simple desk calculator only deals with single digit values.
\$i represents the value associated with the ith grammar symbol in the rhs.
\$\$ is the value returned.

YACC Specification for a More Advanced Desk Calculator

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* double type for Yacc stack */
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines   :   lines expr '\n'   { printf("%g\n", $2); }
        |   lines '\n'
        |   /* e */
        ;
expr    :   expr '+' expr     { $$ = $1 + $3; }
        |   expr '-' expr     { $$ = $1 - $3; }
        |   expr '*' expr     { $$ = $1 * $3; }
        |   expr '/' expr     { $$ = $1 / $3; }
        |   '(' expr ')'      { $$ = $2; }
        |   '-' expr  %prec UMINUS  { $$ = - $2; }
        |   NUMBER
        ;
%%
yylex() {
    int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c == '.') || (isdigit(c)) ) {
       ungetc(c, stdin);
       scanf("%lf", &yylval);
       return NUMBER;
    }
    return c;
}
```

Fig. 4.59. Yacc specification for a more advanced desk calculator.

Note there is only one nonterminal and it is a type double.
So defining YYSTYPE is one way of defining a type for the attributes.
There is also a \%union{ ... } command to define multiple types, where each terminal and nonterminal symbol would be associated with a particular type.
Note the use of the \%left and \%right directives to resolve the conflicts.
Note the UMINUS that just serves as a higher precedence marker.
Note the more advanced yylex() routine to read in a double precision value.
\textit{yylval} is the value returned associated with a token from yylex().

Desk Calculator with Error Recovery

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* double type for Yacc stack */
%}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
lines   :  lines expr '\n'   { printf("%g\n", $2); }
        |  lines '\n'
        |  /* empty */
        |  error '\n'{ yyerror("reenter last line:");
                       yyerrok; }
        ;
expr    :  expr '+' expr     { $$ = $1 + $3; }
        |  expr '-' expr     { $$ = $1 - $3; }
        |  expr '*' expr     { $$ = $1 * $3; }
        |  expr '/' expr     { $$ = $1 / $3; }
        |  '(' expr ')'      { $$ = $2; }
        |  '-' expr  %prec UMINUS  { $$ = - $2; }
        |  NUMBER
        ;
%%
#include "lex.yy.c"
```

Fig. 4.41. Desk calculator with error recovery.

The token \textit{error} is reserved for error handling.
Note the error production that allows the parse to pass, but invokes the appropriate action.
It keeps shifting input symbols until it finds a '\\n'.
\textit{yyerror()} just prints a message to standard error.
The routine \textit{yyerrok()} resets the parser to its normal mode of operation.
In other words, the parser is no longer in an error state, the line can be reentered and the next input symbol will be considered part of the string.