Syntax Analysis

Syntax Analysis

. Check syntax and construct abstract syntax tree

. Error reporting and recovery

. Model using context free grammars

. Recognize using Push down automata/Table Driven Parsers

This is the second phase of the compiler. In this phase, we check the syntax and construct the abstract syntax tree. This phase is modeled through context free grammars and the structure is recognized through push down automata or table-driven parsers. The syntax analysis phase verifies that the string can be generated by the grammar for the source language. In case of any syntax errors in the program, the parser tries to report as many errors as possible. Error reporting and recovery form a very important part of the syntax analyzer. The error handler in the parser has the following goals: . It should report the presence of errors clearly and accurately. . It should recover from each error quickly enough to be able to detect subsequent errors. . It should not significantly slow down the processing of correct programs.

What syntax analysis cannot do!

. To check whether variables are of types on which operations are allowed

. To check whether a variable has been declared before use

. To check whether a variable has been initialized

. These issues will be handled in semantic analysis

The information which syntax analysis phase gets from the previous phase (lexical analysis) is whether a token is valid or not and which class of tokens does it belong to. Hence it is beyond the capabilities of the syntax analysis phase to settle issues like:

. Whether or not a variable has already been declared?

. Whether or not a variable has been initialized before use?

. Whether or not a variable is of the type on which the operation is allowed?

All such issues are handled in the semantic analysis phase.

Limitations of regular languages

. How to describe language syntax precisely and conveniently. Can regular expressions be used?

. Many languages are not regular, for example, string of balanced parentheses

- ((((.))))
- { (i) i | i = 0 }
- There is no regular expression for this language

. A finite automata may repeat states, however, it cannot remember the number of times it has been to a particular state

. A more powerful language is needed to describe a valid string of tokens

Regular expressions cannot be used to describe language syntax precisely and conveniently. There are many languages which are not regular. For example, consider a language consisting of all strings of balanced parentheses. There is no regular expression for this language. Regular expressions can not be used for syntax analysis (specification of grammar) because: . The pumping lemma for regular languages prevents the representation of constructs like a string of balanced parenthesis where there is no limit on the number of parenthesis. Such constructs are

allowed by most of the programming languages. . This is because a finite automaton may repeat states, however, it does not have the power to remember the number of times a state has been reached. . Many programming languages have an inherently recursive structure that can be defined by Context Free Grammars (CFG) rather intuitively. So a more powerful language is needed to describe valid string of tokens.

Syntax definition

. Context free grammars

- a set of tokens (terminal symbols)
- a set of non terminal symbols
- a set of productions of the form nonterminal String of terminals & non terminals
- a start symbol <T, N, P, S>

. A grammar derives strings by beginning with a start symbol and repeatedly replacing a non terminal by the right hand side of a production for that non terminal.

. The strings that can be derived from the start symbol of a grammar G form the language L(G) defined by the grammar.

A context free grammar has four components:

- **A set of tokens , known as terminal symbols. Terminals are the basic symbols from which strings are formed.**
- **A set of non-terminals . Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.**
- **A set of productions . The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.**
- **A designation of one of the non-terminals as the start symbol , and the set of strings it denotes is the language defined by the grammar.**

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right hand side of a production for that non-terminal.

Examples

. String of balanced parentheses

S →(S) S | **ε**

. Grammar

list →list + digit | list - digit | digit

digit→ 0 | 1 | . | 9 Consists of the language which is a list of digit separated by + or -.

S→ (S) S | ε

is the grammar for a string of balanced parentheses.

For example, consider the string: (( )( )). It can be derived as:

S → (S)S → ((S)S)S → (( )S)S→ (( )(S)S)S → (( )( )S)S → (( )( ))S → (( )( ))

Similarly,

list →list + digit

| list - digit

| digit digit 0 | 1 | . | 9

is the grammar for a string of digits separated by + or -.

Derivation

list → list + digit

list - digit + digit
digit - digit + digit
9 - digit + digit
9 - 5 + digit
9 - 5 + 2

Therefore, the string 9-5+2 belongs to the language specified by the grammar.

The name context free comes from the fact that use of a production X. does not depend on the context of X

For example, consider the string 9 - 5 + 2 . It can be derived as:

list →list + digit →list - digit + digit→digit - digit + digit →9 - digit + digit→9 - 5 + digit→9 - 5 + 2

It would be interesting to know that the name context free grammar comes from the fact that use of a production X. does not depend on the context of X.

Examples .

. Grammar for Pascal block

block begin statements end
statements stmt-list | ε
stmt-list stmt-list ; stmt
| stmt

block begin statements end
statements stmt-list | ε
stmt-list stmt-list ; stmt
| stmt

is the grammar for a block of Pascal language.

Syntax analyzers

. Testing for membership whether w belongs to L(G) is just a "yes" or "no" answer
. However the syntax analyzer
- Must generate the parse tree
- Handle errors gracefully if string is not in the language

. Form of the grammar is important
- Many grammars generate the same language
- Tools are sensitive to the grammar

A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order. Each interior node of a parse tree is labeled by some non-terminal **A** , and that the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this **A** was replaced in the derivation. A syntax analyzer not only tests whether a construct is syntactically correct i.e. belongs to the language represented by the specified grammar but also generates the parse tree. It also reports appropriate error messages in case the string is not in the language represented by the grammar specified. It is possible that many grammars represent the same language. However, the tools such as ***yacc*** or other **parser generators** are sensitive to the grammar form. For example, if the grammar has shift-shift or shift-reduce conflicts, the parser tool will give appropriate warning message. We will study about these in details in the subsequent sections.

Derivation

. If there is a production A →a then we say that A derives a and is denoted by A $\Rightarrow$a
. a A ß $\Rightarrow$a γ ß if A γ is a production
. If $a_1 \Rightarrow a_2 \Rightarrow . \Rightarrow a_n$ then a $_1\Rightarrow a_n$

. Given a grammar G and a string w of terminals in L(G) we can write S ⇒ w

. If S ⇒ a where a is a string of terminals and non terminals of G then we say that a is a sentential form of G

If there is a production A →a then it is read as " A derives a " and is denoted by A ⇒ a . The production tells us that we could replace one instance of an A in any string of grammar symbols by a .

In a more abstract setting, we say that a A ß ⇒ a γ ß if A ? is a production and a and ß are arbitrary strings of grammar symbols

If a 1 ⇒ a 2 ⇒ . ⇒ a n then we say a 1 derives a n . The symbol ⇒ means "derives in one step". Often we wish to say "derives in one or more steps". For this purpose, we can use the symbol ⇒ with a + on its top as shown in the slide. Thus, if a string w of terminals belongs to a grammar G, it

is written as $S \overset{+}{\Rightarrow} w$ **. If** $S \overset{*}{\Rightarrow} a$ **, where a may contain non-terminals, then we say that a is a sentential form of G. A sentence is a sentential form with no non-terminals.**

Derivation .

. If in a sentential form only the leftmost non terminal is replaced then it becomes leftmost derivation

. Every leftmost step can be written as wAγ $\overset{lm*}{\Rightarrow}$ wδγ where w is a string of terminals and A δ is a production

. Similarly, right most derivation can be defined

. An ambiguous grammar is one that produces more than one leftmost/rightmost derivation of a sentence

Consider the derivations in which only the leftmost non-terminal in any sentential form is replaced at each step. Such derivations are termed leftmost derivations. If **a** ⇒ **ß** by a step in which the leftmost non-terminal in **a** is replaced, we write **a** $\overset{lm}{\Rightarrow}$ **ß** . Using our notational conventions, every leftmost step can be written **wAγ** $\overset{lm}{\Rightarrow}$ **wdγ** where **w** consists of terminals only, **A d i**s the production applied, and **? is a string of grammar symbols. If a derives ß by a leftmost derivation, then we write a** $\overset{lm*}{\Rightarrow}$ **ß** . If **S** $\overset{lm*}{\Rightarrow}$ **a** , then we say **a** is a left-sentential form of the grammar at hand. Analogous definitions hold for rightmost derivations in which the rightmost non-terminal is replaced at each step. Rightmost derivations are sometimes called the canonical derivations. A grammar that produces more than one leftmost or more than one rightmost derivation for some sentence is said to be **ambiguous** . Put another way, an ambiguous grammar is one that produces more than one parse tree for some sentence is said to be ambiguous.

Parse tree

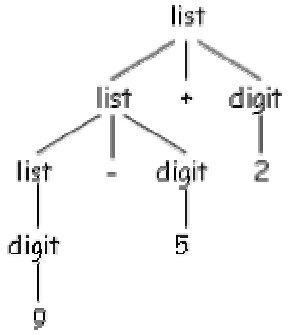
- It shows how the start symbol of a grammar derives a string in the language
- root is labeled by the start symbol
- leaf nodes are labeled by tokens
- Each internal node is labeled by a non terminal
- if A is a non-terminal labeling an internal node and x_1 , x_2 , .x_n are labels of the children of that node then A x_1 x_2 . x_n is a production

A parse tree may be viewed as a graphical representation for a derivation that filters out the choice regarding replacement order. Thus, a parse tree pictorially shows how the start symbol of a grammar derives a string in the language. Each interior node of a parse tree is labeled by some non-terminal A , and that the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this A was replaced in the derivation. The root of the parse tree is labeled by the start symbol and the leaves by non-terminals or terminals and, read from left to right, they constitute a sentential form, called the yield or frontier of the tree. So, if A

is a non-terminal labeling an internal node and x_1 , x_2 , .x_n are labels of children of that node then A →x_1 x_2 . x_n is a production. We will consider an example in the next slide.

Example

Parse tree for 9-5+2



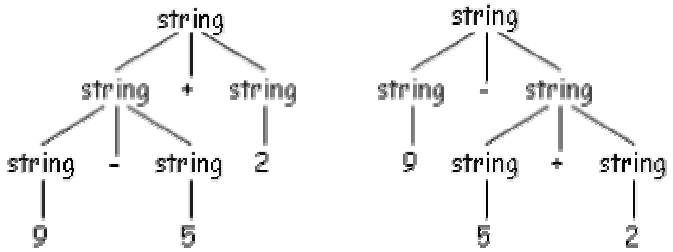
The parse tree for 9-5+2 implied by the derivation in one of the previous slides is shown.
. 9 is a *list* by production (3), since 9 is a digit.
. 9-5 is a *list* by production (2), since 9 is a list and 5 is a digit.
. 9-5+2 is a *list* by production (1), since 9-5 is a list and 2 is a digit.
Production 1: list list + digit
Production 2: list list - digit
Production 3: list digit
digit 0|1|2|3|4|5|6|7|8|9

Ambiguity

. A Grammar can have more than one parse tree for a string
Consider grammar
string →string + string
| string - string
| 0 | 1 | . | 9
. String 9-5+2 has two parse trees

A grammar is said to be an ambiguous grammar if there is some string that it can generate in more than one way (i.e., the string has more than one parse tree or more than one leftmost derivation). A language is inherently ambiguous if it can only be generated by ambiguous grammars.

For example, consider the following grammar:
string →string + string
| string - string
| 0 | 1 | . | 9
In this grammar, the string 9-5+2 has two possible parse trees as shown:



Consider the parse trees for string 9-5+2, expression like this has more than one parse tree. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5)+2 and 9-(5+2). The second parenthesization gives the expression the value 2 instead of 6.

Ambiguity .

. Ambiguity is problematic because meaning of the programs can be incorrect
. Ambiguity can be handled in several ways
- Enforce associativity and precedence
- Rewrite the grammar (cleanest way)

. There are no general techniques for handling ambiguity
. It is impossible to convert automatically an ambiguous grammar to an unambiguous one

Ambiguity is harmful to the intent of the program. The input might be deciphered in a way which was not really the intention of the programmer, as shown above in the 9-5+2 example. Though there is no general technique to handle ambiguity i.e., it is not possible to develop some feature which automatically identifies and removes ambiguity from any grammar. However, it can be removed, broadly speaking, in the following possible ways:-

1) Rewriting the whole grammar unambiguously.
2) Implementing precedence and associatively rules in the grammar. We shall discuss this technique in the later slides.

Associativity

. If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator
. In a+b+c b is taken by left +
. +, -, *, / are left associative
. ^, = are right associative
. Grammar to generate strings with right associative operators right à letter = right | letter letter
a| b |.| z

A binary operation * on a set S that does not satisfy the associative law is called non-associative.
A left-associative operation is a non-associative operation that is conventionally evaluated from left to right i.e., operand is taken by the operator on the left side.
For example,
6*5*4 = (6*5)*4 and not 6*(5*4)
6/5/4 = (6/5)/4 and not 6/(5/4)
A right-associative operation is a non-associative operation that is conventionally evaluated from right to left i.e., operand is taken by the operator on the right side.
For example,
6^5^4 => 6^(5^4) and not (6^5)^4)
x=y=z=5 => x=(y=(z=5))
Following is the grammar to generate strings with left associative operators. (Note that this is left recursive and may go into infinite loop. But we will handle this problem later on by making it right recursive)
left → left + letter | letter
letter→ a | b | .... | z

Precedence

. String a+5*2 has two possible interpretations because of two different parse trees corresponding to
(a+5)*2 and a+(5*2)
. Precedence determines the correct interpretation.

Precedence is a simple ordering, based on either importance or sequence. One thing is said to "take precedence" over another if it is either regarded as more important or is to be performed first. For

example, consider the string a+5*2. It has two possible interpretations because of two different parse trees corresponding to (a+5)*2 and a+(5*2). But the * operator has precedence over the + operator. So, the second interpretation is correct. Hence, the precedence determines the correct interpretation.

Parsing

. Process of determination whether a string can be generated by a grammar

. Parsing falls in two categories:

- Top-down parsing: Construction of the parse tree starts at the root (from the start symbol) and proceeds towards leaves (token or terminals)
- Bottom-up parsing: Construction of the parse tree starts from the leaf nodes (tokens or terminals of the grammar) and proceeds towards root (start symbol)

Parsing is the process of analyzing a continuous stream of input (read from a file or a keyboard, for example) in order to determine its grammatical structure with respect to a given formal grammar. The task of the parser is essentially to determine if and how the input can be derived from the start symbol within the rules of the formal grammar. This can be done in essentially two ways:

. Top-down parsing - A parser can start with the start symbol and try to transform it to the input. Intuitively, the parser starts from the largest elements and breaks them down into incrementally smaller parts. LL parsers are examples of top-down parsers. We will study about these in detail in the coming slides.

. Bottom-up parsing - A parser can start with the input and attempt to rewrite it to the start symbol. Intuitively, the parser attempts to locate the most basic elements, then the elements containing these, and so on. LR parsers are examples of bottom-up parsers.

Example: Top down Parsing

. Following grammar generates types of Pascal

type →simple
| ↑id
| array [simple] of type

simple →integer
| char
| num dotdot num

Top-down parsing is a strategy of analyzing unknown data relationships by hypothesizing general parse tree structures and then considering whether the known fundamental structures are compatible with the hypothesis. For example, the following grammar generates the types in Pascal language by starting from *type* and generating the string:

type →simple
| ↑id
| array [simple] of type

simple →integer
| char
| num dotdot num

Example .

. Construction of a parse tree is done by starting the root labeled by a start symbol

. repeat following two steps

- at a node labeled with non terminal A select one of the productions of A and construct children nodes
- find the next node at which subtree is Constructed

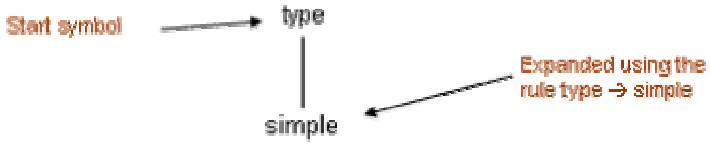
To construct a parse tree for a string, we initially create a tree consisting of a single node (root node) labeled by the start symbol. Thereafter, we repeat the following steps to construct the of parse tree by starting at the root labeled by start symbol:

. At node labeled with non terminal A select one of the production of A and construct the children nodes.

. Find the next node at which subtree is constructed.

Example .

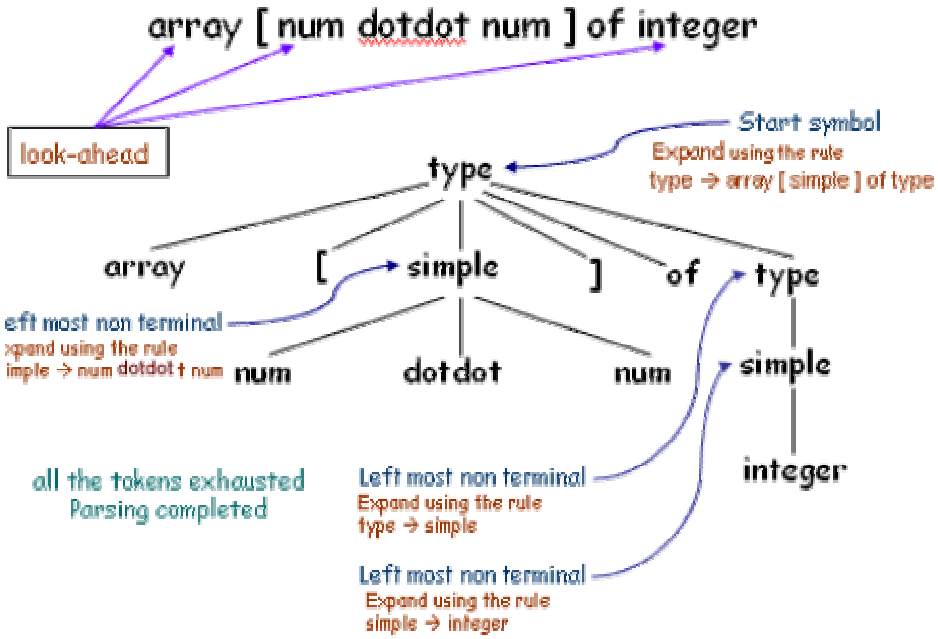
. Parse array [num dotdot num] of integer



. Cannot proceed as non terminal "simple" never generates a string beginning with token "array". Therefore, requires back-tracking.

. Back-tracking is not desirable, therefore, take help of a "look-ahead" token. The current token is treated as look- ahead token. (restricts the class of grammars)

To construct a parse tree corresponding to the string *array [num dotdot num] of integer* , we start with the start symbol *type* . Then, we use the production type à simple to expand the tree further and construct the first child node. Now, finally, the non-terminal *simple* should lead to the original string. But, as we can see from the grammar, the expansion of the non-terminal *simple* never generates a string beginning with the token "array". So, at this stage, we come to know that we had used the wrong production to expand the tree in the first step and we should have used some other production. So, we need to backtrack now. This backtracking tends to cause a lot of overhead during the parsing of a string and is therefore not desirable. To overcome this problem, a " look-ahead " token can be used. In this method, the current token is treated as look-ahead token and the parse tree is expanded by using the production which is determined with the help of the look-ahead token.



Parse *array [num dotdot num] of integer* using the grammar:
type →simple
| ↑id |
array [simple] of type
simple→ integer | char | num dotdot num
Initially, the token *array* is the lookahead symbol and the known part of the parse tree consists of the root, labeled with the starting non- terminal *type.* For a match to occur, non-terminal *type* must derive a string that starts with the lookahead symbol *array* . In the grammar, there is just one production of such type, so we select it, and construct the children of the root labeled with the right side of the production. In this way we continue, when the node being considered on the parse tree is for a terminal and the terminal matches the lookahead symbol, then we advance in both the parse tree and the input. The next token in the input becomes the new lookahead symbol and the next child in the parse tree is considered.

Recursive descent parsing

First set:
Let there be a production
A →α
then First(α) is the set of tokens that appear as the first token in the strings generated from α
For example :
First(simple) = {integer, char, num}
First(num dotdot num) = {num}
Recursive descent parsing is a top down method of syntax analysis in which a set of recursive procedures are executed to process the input. A procedure is associated with each non-terminal of the grammar. Thus, a recursive descent parser is a top-down parser built from a set of mutually-recursive procedures or a non-recursive equivalent where each such procedure usually implements one of the production rules of the grammar.

For example, consider the grammar,

type →simple | ↑id | array [simple] of type

simple →integer | char | num dot dot num

First (a) **is the set of terminals that begin the strings derived from a . If a derives e then e too is in First (a). This set is called the first set of the symbol a . Therefore,**

First (simple) = {integer, char , num}

First (num dot dot num) = {num}

First (type) = {integer, char, num, ↑, array}

Define a procedure for each non terminal

```
procedure type;
  if lookahead in {integer, char, num}
   then simple
   else if lookahead = ↑
          then begin match( ↑ );
                     match(id)
               end
          else if lookahead = array
                 then begin match(array);
                            match( [ );
                            simple;
                            match( ] );
                            match(of);
                            type
                      end
                 else error;
```

Procedure for each of the non-terminal is shown.

```
procedure simple;
  if lookahead = integer
    then match(integer)
    else if lookahead = char
           then match(char)
           else if lookahead = num
                  then begin match(num);
                             match(dotdot);
                             match(num)
                       end
                  else
                       error;

procedure match(t:token);
   if lookahead = t
      then lookahead = next token
      else error;
```

Apart from a procedure for each non-terminal we also needed an additional procedure named *match* . *match* advances to the next input token if its argument t matches the lookahead symbol.

Ambiguity

. Dangling else problem

Stmt →if expr then stmt

| if expr then stmt else stmt

. according to this grammar, string if el then if e2 then S1 else S2 has two parse trees

The dangling else is a well-known problem in computer programming in which a seemingly well-defined grammar can become ambiguous. In many programming languages you can write code like *if a then if b then s1 else s2* which can be understood in two ways:

Either as

if a then

if b then

s1

else

s2

or as

if a then

if b then

s1

else

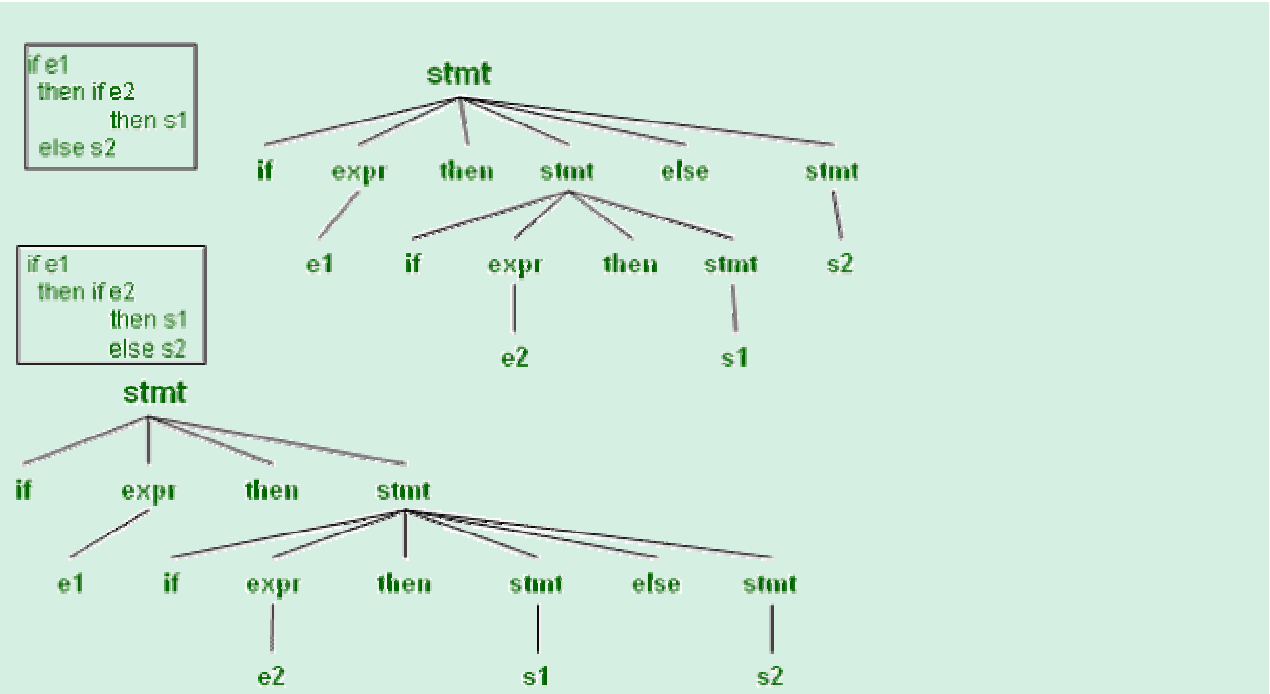
s2

So, according to the following grammar, the string *if el then if e2 then S1 else S2* will have two parse trees as shown in the next slide.

stmt →if expr then stmt

| if expr then stmt else stmt

| other



The two parse trees corresponding to the string *if a then if b then s1 else s2* .

Resolving dangling else problem

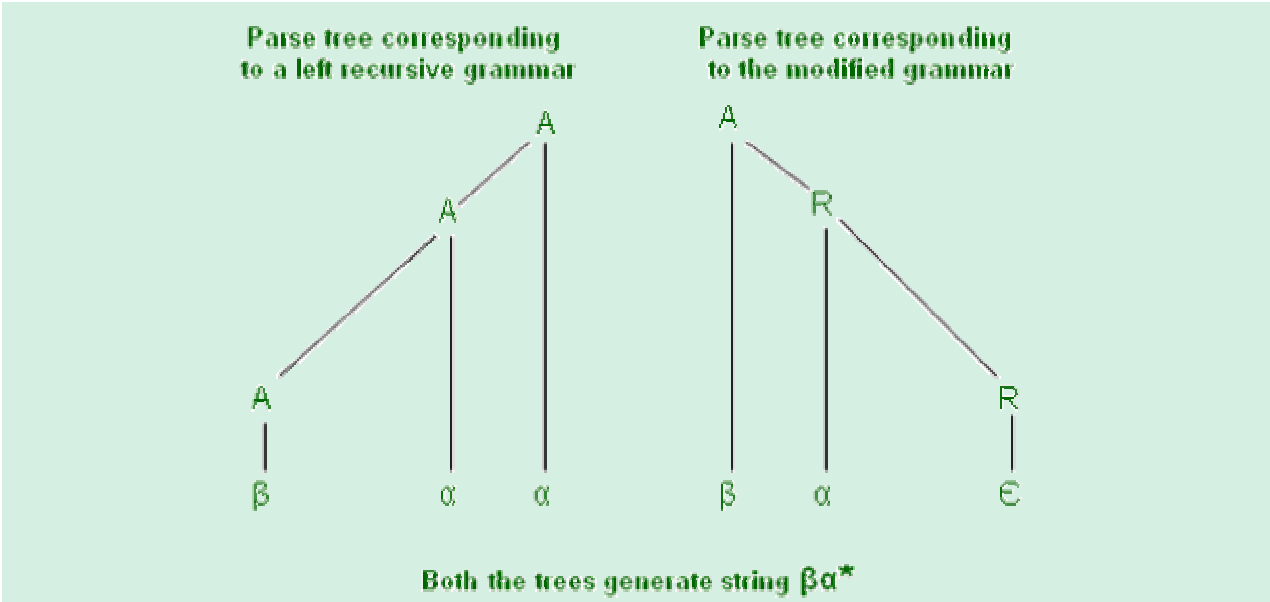
- General rule: match each **else** with the closest previous **then.** The grammar can be rewritten as
  stmt →matched-stmt
  | unmatched-stmt
  | others
  matched-stmt →if expr then matched-stmt
  else matched-stmt
  | others

So, we need to have some way to decide to which if an ambiguous else should be associated. It can be solved either at the implementation level, by telling the parser what the right way to solve the ambiguity, or at the grammar level by using a Parsing expression grammar or equivalent. Basically, the idea is that a statement appearing between a *then* and an *else* must be matched i.e., it must not end with an unmatched *then* followed by any statement, for the *else* would then be forced to match this unmatched *then* . So, the general rule is " *Match each else with the closest previous unmatched then"*.

Thus, we can rewrite the grammar as the following unambiguous grammar to eliminate the dangling else problem:
stmt →matched-stmt | unmatched-stmt | others
matched-stmt → if expr then matched-stmt else matched-stmt | others
unmatched-stmt →if expr then stmt
| if expr then matched-stmt else unmatched-stmt
A matched statement is either an if-then-else statement containing no unmatched statements or it is any other kind of unconditional statement.

Left recursion

. A top down parser with production A →A α may loop forever
. From the grammar A →A α | b left recursion may be eliminated by transforming the grammar to
A →b R
R →α R | ε
Left recursion is an issue of concern in top down parsers. A grammar is left-recursive if we can find some non-terminal A which will eventually derive a sentential form with itself as the left-symbol. In other words, a grammar is *left recursive* if it has a non terminal A such that there is a derivation
A →+ A a for some string a . These derivations may lead to an infinite loop. Removal of left recursion:
For the grammar A →A a | ß , left recursion can be eliminated by transforming the original grammar as:
A →ß R
R →a R | ε



This slide shows the parse trees corresponding to the string ßa * using the original grammar (with left factoring) and the modified grammar (without left factoring).

Example

. Consider grammar for arithmetic expressions

E →E + T | T

T →T * F | F

F → (E) | id

. After removal of left recursion the grammar becomes

E →T E'

E' →+ T E' | ε

T →F T'

T' →* F T' | ε

F → (E) | id

As another example, a grammar having left recursion and its modified version with left recursion removed has been shown.

Removal of left recursion

In general

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots\ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$$

transforms to

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots\ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots\ldots \mid \alpha_m A' \mid Є$$

The general algorithm to remove the left recursion follows. Several improvements to this method have been made. For each rule of the form

$A \rightarrow A\,\mathbf{a}_1 \mid A\,\mathbf{a}_2 \mid ... \mid A\,\mathbf{a}_m \mid \beta_1 \mid \beta_2 \mid .. \mid \beta_n$

Where:

. **A** is a left-recursive non-terminal.

. **a** is a sequence of non-terminals and terminals that is not null ($a \neq \varepsilon$).

. **ß** is a sequence of non-terminals and terminals that does not start with **A** .

Replace the A-production by the production:

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$

And create a new non-terminal

$A' \rightarrow \mathbf{a}_1 A' \mid \mathbf{a}_2 A' \mid ... \mid \mathbf{a}_m A' \mid \varepsilon$

This newly created symbol is often called the "tail", or the "rest".

Left recursion hidden due to many productions

.Left recursion may also be introduced by two or more grammar rules. For example

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid \varepsilon$

there is a left recursion because

$S \rightarrow Aa \rightarrow Sda$

. In such cases, left recursion is removed systematically

- Starting from the first rule and replacing all the occurrences of the first non terminal symbol
- Removing left recursion from the modified grammar

What we saw earlier was an example of immediate left recursion but there may be subtle cases where left recursion occurs involving two or more productions. For example in the grammar

$S \rightarrow A\,a \mid b$

$A \rightarrow A\,c \mid S\,d \mid e$

there is a left recursion because

$S \rightarrow A\,a \rightarrow S\,d\,a$

More generally, for the non-terminals *A* 0, *A* 1,..., *An* , indirect left recursion can be defined as being of the form:

$A_n \rightarrow A_1\,a_1 \mid .$

$A_1 \rightarrow A_2\,a_2 \mid$

. .

$A_n \rightarrow A_n\,a_{(n+1)} \mid .$

Where a_1 , a_2 ,..., a_n are sequences of non-terminals and terminals.

Following algorithm may be used for removal of left recursion in general case:

Input : Grammar G with no cycles or e -productions.

Output: An equivalent grammar with no left recursion .

Algorithm:

Arrange the non-terminals in some order A_1 , A_2 , A_3 ..... A_n .

for i := 1 to n do begin

replace each production of the form $A_i \rightarrow A_{j\gamma}$

by the productions $A\,i \rightarrow d_1\gamma \mid d_2\gamma \mid ..... \mid d_k\gamma$

where $A_j \rightarrow d_1 \mid d_2 \mid ..... \mid d_k$ are all the current Aj -productions;

end

eliminate the immediate left recursion among the Ai-productions.

end.

Removal of left recursion due to many productions .

$S \rightarrow A\,a \mid b$

A → A c | S d | e
. After the first step **(substitute S by its rhs in the rules)** the grammar becomes
S →Aa | b
A →Ac | Aad | bd | ε
. After the second step (removal of left recursion) the grammar becomes
S →Aa | b
A →bdA' | A'
A' →cA' | adA' | ε
After the first step (substitute S by its R.H.S. in the rules), the grammar becomes
S →A a | b
A →A c | A a d | b d | ε
After the second step (removal of left recursion from the modified grammar obtained after the first step), the grammar becomes
S →A a | b
A →b d A' | A'
A' →c A' | a d A' | ε
Left factoring
. In top-down parsing when it is not clear which production to choose for expansion of a symbol
defer the decision till we have seen enough input.
In general if A →$\alpha\beta_1$ | $\alpha\beta_2$
defer decision by expanding A to a A'
we can then expand A' to β_1 or β_2
. Therefore A →$\alpha\beta_1$ | $\alpha\beta_2$
transforms to
A →α A'
A' →β_1 | β_2
Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two or more alternative productions to use to expand a non-terminal A, we defer the decision till we have seen enough input to make the right choice.
In general if A →α ß $_1$ | α ß $_2$, we defer decision by expanding A to a A'.
and then we can expand A ' to $ß_1$ or $ß_1$
Therefore A →α $ß_1$ | α $ß_2$ transforms to
A →α A'
A ' →$ß_1$ | $ß_2$
Dangling else problem again
Dangling else problem can be handled by left factoring
stmt →if expr then stmt else stmt
| if expr then stmt
can be transformed to
stmt →if expr then stmt S'
S' →else stmt | ε
We can also take care of the dangling else problem by left factoring. This can be done by left factoring the original grammar thus transforming it to the left factored form.
stmt →if expr then stmt else stmt
| if expr then stmt
is transformed to
stmt →if expr then stmt S'

S' →else stmt | ε

Predictive parsers

. A non recursive top down parsing method

. **Parser "predicts" which production to use**

. **It removes backtracking by fixing one production for every non-terminal and input token(s)**

. **Predictive parsers accept LL(k) languages**

- First L stands for left to right scan of input
- Second L stands for leftmost derivation
- k stands for number of lookahead token

. **In practice LL(1) is used**

In general, the selection of a production for a non-terminal may involve trial-and-error; that is, we may have to try a production and backtrack to try another production if the first is found to be unsuitable. A production is unsuitable if, after using the production, we cannot complete the tree to match the input string. Predictive parsing is a special form of recursive-descent parsing, in which the current input token unambiguously determines the production to be applied at each step. After eliminating left recursion and left factoring, we can obtain a grammar that can be parsed by a recursive-descent parser that needs no *backtracking* . Basically, it removes the need of *backtracking* by fixing one production for every non-terminal and input tokens. Predictive parsers accept LL(k) languages where:

. First **L** : The input is scanned from left to right.

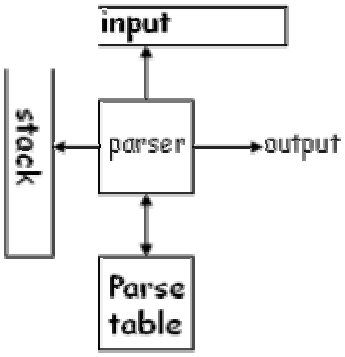
. Second **L** : Leftmost derivations are derived for the strings.

. **k** : The number of lookahead tokens is k.

However, in practice, LL(1) grammars are used i.e., one lookahead token is used.

Predictive parsing

. Predictive parser can be implemented by maintaining an external stack



Parse table is a two dimensional array M[X,a] where "X" is a non terminal and "a" is a terminal of the grammar

It is possible to build a non recursive predictive parser maintaining a stack explicitly, rather than implicitly via recursive calls. A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string. The stack contains a sequence of grammar symbols with a $ on the bottom, indicating the bottom of the stack. Initially the stack contains the start symbol of the grammar on top of $. The parsing table is a two-dimensional array M [X,a] , where X is a non-terminal, and a is a terminal or the symbol $. The key problem during predictive parsing is that of determining the production to be applied for a non-terminal. The non-recursive parser looks up the production to be applied in the parsing table. We shall see how a predictive parser works in the subsequent slides.

Parsing algorithm

. The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol

. These two symbols determine the action to be taken by the parser

. Assume that '$' is a special token that is at the bottom of the stack and terminates the input string

if X = a = $ then halt

if X = a ≠ $ then pop(x) and ip++

if X is a non terminal

then if M[X,a] = {X à UVW}

then begin pop(X); push(W,V,U)

end

else error

The parser is controlled by a program that behaves as follows. The program considers X , the symbol on top of the stack, and a , the current input symbol. These two symbols determine the action of the parser. Let us assume that a special symbol ' $ ' is at the bottom of the stack and terminates the input string. There are three possibilities:

1. If X = a = $, the parser halts and announces successful completion of parsing.
2. If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a] = {X UVW}, the parser replaces X on top of the stack by UVW (with U on the top). If M[X,a] = error, the parser calls an error recovery routine.

The behavior of the parser can be described in terms of its configurations, which give the stack contents and the remaining input.

Example

. Consider the grammar

E →T E'

E' →+T E' | ε

T →F T'

T' →* F T' | ε

F → (E) | id

As an example, we shall consider the grammar shown. A predictive parsing table for this grammar is shown in the next slide. We shall see how to construct this table later.

Parse table for the grammar

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ϵ	E'→ϵ
T	T→FT'			T→FT'		
T'		T'→ϵ	T'→*FT'		T'→ϵ	T'→ϵ
F	F→id			F→(E)		

Blank entries are error states. For example E cannot derive a string starting with '+'

A predictive parsing table for the grammar in the previous slide is shown. In the table, the blank entries denote the error states; non-blanks indicate a production with which to expand the top nonterminal on the stack.

Example

Stack	input	action
$E	id + id * id $	expand by E TE'
$E'T	id + id * id $	expand by T FT'
$E'T'F	id + id * id $	expand by F id
$E'T'id	id + id * id $	pop id and ip++
$E'T'	+ id * id $	expand by T' ε
$E'	+ id * id $	expand by E' +TE'
$E'T+	+ id * id $	pop + and ip++
$E'T	id * id $	expand by T FT'

Let us work out an example assuming that we have a parse table. We follow the predictive parsing algorithm which was stated a few slides ago. With input **id id * id** , the predictive parser makes the sequence of moves as shown. The input pointer points to the leftmost symbol of the string in the INPUT column. If we observe the actions of this parser carefully, we see that it is tracing out a leftmost derivation for the input, that is, the productions output are those of a leftmost derivation. The input symbols that have already been scanned, followed by the grammar symbols on the stack (from the top to bottom), make up the left-sentential forms in the derivation.

Constructing parse table

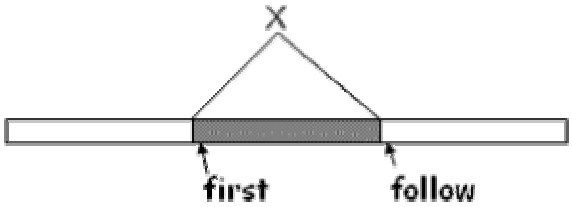
. Table can be constructed if for every non terminal, every lookahead symbol can be handled by at most one production

. First(a) for a string of terminals and non terminals a is

- Set of symbols that might begin the fully expanded (made of only tokens) version of a

. Follow(X) for a non terminal X is

- set of symbols that might follow the derivation of X in the input stream



The construction of the parse table is aided by two functions associated with a grammar G. These functions, **FIRST** and **FOLLOW** , allow us to fill in the entries of a predictive parsing table for G, whenever possible. If α is any string of grammar symbols, **FIRST** (**α**) is the set of terminals that begin the strings derived from **α** . If α $\Rightarrow$* ε , then ε is also in FIRST(α).

If **X** is a non-terminal, **FOLLOW** (**X**) is the set of terminals *a* that can appear immediately to the right of **X** in some sentential form, that is, the set of terminals **a** such that there exists a derivation of the form **S** $\Rightarrow$* **α A *a* ß** for some **α** and **ß** . Note that there may, at some time, during the derivation, have been symbols between **A** and **a** , but if so, they derived **ε** and disappeared. Also, if **A** can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

Compute first sets

. If X is a terminal symbol then First(X) = {X}
. If X →ε is a production then ε is in First(X)
. If X is a non terminal
and X →Y_1 Y_2 2 . Y_k is a production
then
if for some i, a is in First(Y_i)
and ε is in all of First(Y_j) (such that j<i)
then a is in First(X)
. If ε is in First (Y_1) . First(Y_k) then ε is in First(X)
To compute FIRST (X) for all grammar symbols X, apply the following rules until no more terminals or e can be added to any FIRST set.
1. If X is terminal, then First (X) is {X}.
2. If X → ε is a production then add e to FIRST(X).
3. If X is a non terminal and X →Y_1 Y_k .....Y_k is a production, then place *a* in First (X) if for some i, *a* is in FIRST(Y_i) and e is in all of FIRST(Y_1), FIRST(Y_2),.., FIRST(Y_{i-1});that is, Y_1 ..Y_{i-1} $\Rightarrow^*$ ε . If ε is in FIRST(Y_j) for all i = 1,2,..,k, then add ε to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ε , then we add nothing more to FIRST(X), but if Y_1 $\Rightarrow^*$ ε , then we add FIRST(Y_2) and so on.

Example

. For the expression grammar
E →T E'
E' →+T E' | ε
T →F T'
T' →* F T' | ε
F → (E) | id
First(E) = First(T) = First(F) = { (, id }
First(E') = {+, ε }
First(T') = { *, ε }
Consider the grammar shown above. For example, id and left parenthesis are added to FIRST(F) by rule (3) in the definition of FIRST with i = 1 in each case, since FIRST(id) = {id} and FIRST{'('} = { (} by rule (1). Then by rule (3) with i = 1, the production T = FT' implies that id and left parenthesis are in FIRST(T) as well. As another example, e is in FIRST(E') by rule (2).

Compute follow sets

1. Place $ in follow(S)
2. If there is a production A →a B ß then everything in first(ß) (except ε) is in follow(B)
3. If there is a production A →a B then everything in follow(A) is in follow(B)
4. If there is a production A →a B ß and First(ß) contains e then everything in follow(A) is in follow(B)
Since follow sets are defined in terms of follow sets last two steps have to be repeated until follow sets converge
To compute FOLLOW (A) for all non-terminals A , apply the following rules until nothing can be added to any FOLLOW set:
1. Place $ in FOLLOW(S), where S is the start symbol and $ is the input right endmarker.

2. If there is a production A →a Bß, then everything in FIRST(ß) except for e is placed in FOLLOW(B).

3. If there is a production A →a ß, or a production A →a Bß where FIRST(ß) contains e (i.e., ß⇒ * e), then everything in FOLLOW(A) is in FOLLOW(B).

Error handling

. Stop at the first error and print a message

- Compiler writer friendly
- But not user friendly

. Every reasonable compiler must recover from errors and identify as many errors as possible

. However, multiple error messages due to a single fault must be avoided

. Error recovery methods

- Panic mode
- Phrase level recovery
- Error productions
- Global correction

Error handling and recovery is also one of the important tasks for a compiler. Errors can occur at any stage during the compilation. There are many ways in which errors can be handled. One way is to stop as soon as an error is detected and print an error message. This scheme is easy for the programmer to program but not user friendly. Specifically, a good parser should, on encountering a parsing error, issue an error message and resume parsing in some way, repairing the error if possible. However, multiple error messages due to a single fault are undesirable and tend to cause confusion if displayed. Error recovery is thus a non trivial task. The following error recovery methods are commonly used:

1. Panic Mode
2. Phrase level recovery
3. Error productions
4. Global correction

Panic mode

. Simplest and the most popular method

. Most tools provide for specifying panic mode recovery in the grammar

. When an error is detected

- Discard tokens one at a time until a set of tokens is found whose role is clear
- Skip to the next token that can be placed reliably in the parse tree

Let us discuss each of these methods one by one. Panic mode error recovery is the simplest and most commonly used method. On discovering the error, the parser discards input symbols one at a time until one of the designated set of synchronizing tokens is found. The synchronizing tokens are usually delimiters, such as semicolon or *end* , whose role in the source program is clear. The compiler designer must select the synchronizing tokens appropriate for the source language, of course.

Panic mode .

. Consider following code

```
begin
    a = b + c;
    x = p r ;
    h = x < 0;
```

end;

. The second expression has syntax error

. Panic mode recovery for begin-end block skip ahead to next ';' and try to parse the next expression

. It discards one expression and tries to continue parsing

. May fail if no further ';' is found

Consider the code shown in the example above. As we can see, the second expression has a syntax error. Now panic mode recovery for begin-end block states that in this situation skip ahead until the next ' ; ' is seen and try to parse the following expressions thereafter i.e., simply skip the whole expression statement if there is an error detected. However, this recovery method might fail if no further ' ; ' is found. While panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has the advantage of simplicity and, unlike some other methods to be considered later, it is guaranteed not to go into an infinite loop. In situations where multiple errors in the same statement are rare, this method may be quite adequate.

Phrase level recovery

. Make local correction to the input

. Works only in limited situations

- A common programming error which is easily detected
- For example insert a ";" after closing "}" of a class definition

. Does not work very well!

Phrase level recovery is implemented by filling in the blank entries in the predictive parsing table with pointers to error routines. These routines may change, insert, or delete symbols on the input and issue appropriate error messages. They may also pop from the stack. It basically makes local corrections to the input; that is, it may replace a prefix of the remaining input by some string that allows the parser to continue. A typical local correction would be to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon. This type of replacement can correct any input string and has been used in several error-repairing compilers. However, its major drawback is the difficulty it has in coping with situations in which the actual error has occurred before the point of detection.

Error productions

. Add erroneous constructs as productions in the grammar

. Works only for most common mistakes which can be easily identified

. Essentially makes common errors as part of the grammar

. Complicates the grammar and does not work very well

If we have a good idea of the common errors that might be encountered, we can augment the grammar for the language at hand with productions that generate the erroneous constructs. We then use the grammar augmented by these error productions to construct a parser. If an error production is used by a parser, we can generate appropriate error diagnostics to indicate the error construct that has been recognized in the input. The main drawback of this approach is that it tends to complicate the grammar and thus does not work very well.

Global corrections

. Considering the program as a whole find a correct "nearby" program

. Nearness may be measured using certain metric

. PL/C compiler implemented this scheme: anything could be compiled!

. It is complicated and not a very good idea!

Ideally, we would like a compiler to make as few changes as possible in processing the incorrect input string. There are algorithms for choosing a minimal sequence of changes to obtain globally least-cost correction. Given an incorrect input string x and a grammar G, these algorithms will find a parse tree for a related string y such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest

Error Recovery in LL(1) parser

. Error occurs when a parse table entry M[A,a] is empty

. Skip symbols in the input until a token in a selected set (synch) appears

. Place symbols in follow(A) in synch set. Skip tokens until an element in follow(A) is seen. Pop(A) and continue parsing

. Add symbol in first(A) in synch set. Then it may be possible to resume parsing according to A if a symbol in first(A) appears in input.

Let us consider error recovery in an LL(1) parser by panic-mode recovery method. An error occurs when the terminal on top of the stack does not match the next input symbol or when non-terminal A is on top of the stack, ***a*** is the next input symbol, and the parsing table entry **M[A, *a*]** is empty. Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice. Some heuristics are as follows:

1. As a starting point, we can place all symbols in FOLLOW(A) into the synchronizing set for non-terminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.

2. If we add symbols in FIRST(A) to the synchronizing set for non-terminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

Bottom up parsing

. Construct a parse tree for an input string beginning at leaves and going towards root OR
. Reduce a string w of input to start symbol of grammar

Consider a grammar

S →aABe
A →Abc | b
B →d

And reduction of a string

a b b c d e
a A b c d e
a A d e
a A B e
S

Right most derivation

S a A B e
a A d e
a A b c d e
a b b c d e

We will now study a general style of bottom up parsing, also known as shift-reduce parsing. Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). We can think of the process as one of "reducing" a string w to the start symbol of a grammar. At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production, and if the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse. For example, consider the grammar

S→aABe
A →Abc | b
B →d

The sentence a b b c d e can be reduced to S by the following steps:

a b b c d e
a A b c d e
a A d e
a A B e
S

These reductions trace out the following right-most derivation in reverse:

S→ a A B e
a A d e
a A b c d e
a b b c d e

Shift reduce parsing

. Split string being parsed into two parts
- Two parts are separated by a special character "."
- Left part is a string of terminals and non terminals
- Right part is a string of terminals

. Initially the input is .w

A convenient way to implement a shift reduce parser is by using an input buffer to hold the string w to be parsed. At any stage, the input string is split into two parts which are separated by the character ' . '. The left part consists of terminals and non-terminals while the right part consists of terminals only. Initially, the string " .w " is on the input. Shift reduce parsing .

. Bottom up parsing has two actions

. **Shift** : move terminal symbol from right string to left string

if string before shift is a .pqr

then string after shift is a p.qr

.**Reduce** : immediately on the left of "." identify a string same as RHS of a production and replace it by LHS

if string before reduce action is aß .pqr

and A →ß is a production

then string after reduction is a A.pqr

There are two primary operations of a shift-reduce parser namely (1) shift and (2) reduce. In a shift action, the next input symbol is shifted from right string to the left string. For example, if string before shift is " a .pqr " then string after shift would be " a p.qr " . In a reduce action, the parser identifies a string which is same as the RHS of a production and replace it by the non-terminal at LHS. For example, if string before reduce action is " aß .pqr " and A →ß is a production, then string after reduction is " a A.pqr " .

Example

Assume grammar is E →E+E | E*E | id

Parse id*id+id

String	action	
.id*id+id	shift	
id.*id+id	reduce E	id
E.*id+id	shift	
E*.id+id	shift	
E*id.+id	reduce E	id
E*E.+id	reduce E	E*E
E.+id	shift	
E+.id	shift	
E+id.	Reduce E	id
E+E.	Reduce E	E+E
E.	ACCEPT	

Consider the following grammar as an example:

E →E + E

| E * E

| id

Detailed steps through the actions of a shift reduce parser might make in parsing the input string "id * id + id" as shown in the slide.

Shift reduce parsing .

. Symbols on the left of "." are kept on a stack

- Top of the stack is at "."
- Shift pushes a terminal on the stack
- Reduce pops symbols (rhs of production) and pushes a non terminal (lhs of production) onto the stack

. The most important issue: when to shift and when to reduce

. Reduce action should be taken only if the result can be reduced to the start symbol

In actual implementation, a stack is used to hold the grammar symbols that are on the left of " . ". So, basically, the top of the stack is at " . ". The shift operation shifts the next input symbol onto the top of the stack. A reduce operation pops the symbols which are in the RHS of the identified production and pushes the non-terminal (LHS of the production) onto the top of the stack. During the whole procedure, we have to know when to use the shift operation and when to use the reduce operation. We must ensure that a reduce action is taken only if the result of the operation is further reducible to the start symbol. We must explain how choices of action are to be made so the shift reduce parser works correctly.

Bottom up parsing .

. A more powerful parsing technique
. LR grammars - more expensive than LL
. Can handle left recursive grammars
. Can handle virtually all the programming languages
. Natural expression of programming language syntax
. Automatic generation of parsers (Yacc, Bison etc.)
. Detects errors as soon as possible
. Allows better error recovery

Bottom-up parsing is a more powerful parsing technique. Listed are some of the advantages of bottom-up parsing: . A more powerful parsing technique . It is capable of handling almost all the programming languages. . It can fastly handle left recursion in the grammar. . It allows better error recovery by detecting errors as soon as possible.

Issues in bottom up parsing

. How do we know which action to take
    - whether to shift or reduce
    - Which production to use for reduction?
. Sometimes parser can reduce but it should not: X ε can always be reduced!
. Sometimes parser can reduce in different ways!
. Given stack d and input symbol a, should the parser
    - Shift a onto stack (making it d a)
    - Reduce by some production A ß assuming that stack has form αß (making it α A)
    - Stack can have many combinations of αß
    - How to keep track of length of ß ?

    There are several issues in bottom up parsing:
    . Making a decision as to which action (shift or reduce) to take next is an issue. Another question is which production to use for reduction, if a reduction has to be done.
    . If there is a production of the form X→ε , then we can introduce any number of X 's on the left side of the ' . ' (e.g., X and XX can derive the same string). In such cases we may not want the parser to reduce the epsilons to X's.
    . A parser can reduce in different ways. For example, let the grammar be
    S →aABe
    A →Abc | b
    B →d
    then reduction of string " a b b c d e " can be done in the following two ways (3rd line).
    a b b c d e a b b c d e
    a A b c d e a A b c d e
    a A d e a A A c d e

Handle

. Handles always appear at the top of the stack and never inside it

. This makes stack a suitable data structure
. Consider two cases of right most derivation to verify the fact that handle appears on the top of the stack
- S →a Az →aß Byz →aßγyz
- S →a BxAz →a Bxyz →aγxyz
. Bottom up parsing is based on recognizing handles
There are two problems that must be solved if we are to parse by handle pruning. The first is to locate the substring to be reduced in a right sentential form, and the second is to determine what production to choose in case there is more than one production with that substring on the right side. A convenient way to implement a shift-reduce parser is to use a stack to hold grammar symbols and an input buffer to hold the string w to be parsed. We use $ to mark the bottom of the stack and also the right end of the input. Initially, the stack is empty, and the string w is on the input, as follows:

STACK	**INPUT**
$	**w $**

The parser operates by shifting zero or more input symbols onto the stack until a handle ß is on top of the stack. The parser then reduces ß to the left side of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	**INPUT**
$	**w $**

After entering this configuration, the parser halts and announces successful completion of parsing. There is an important fact that justifies the use of a stack in shift-reduce parsing: the handle will always appear on top of the stack, never inside. This fact becomes obvious when we consider the possible forms of two successive steps in any rightmost derivation. These two steps can be of the form:
S →a Az →aß Byz →aßγ yz
S →a BxAz →a Bxyz →aγ xyz
Handle always appears on the top
Case I: S →a Az →aß Byz →aßγ yz

stack	**input**	**action**	
aßγ	yz	reduce by B	γ
aß B	yz	shift y	
aß By	z	reduce by A	ß By
a A	z		

Case II: S →a BxAz →a Bxyz →aγ xyz

stack	**input**	**action**	
aγ	xyz	reduce by B	γ
a B	xyz	shift x	
a Bx	yz	shift y	
a Bxy	z	reduce A	y
a BxA	z		

In case (1), A is replaced by ß By , and then the rightmost non-terminal B in that right side is replaced by γ . In case (2), A is again replaced first, but this time the right side is a string y of terminals only.

The next rightmost non-terminal B will be somewhere to the left of y . In both cases, after making a reduction the parser had to shift zero or more symbols to get the next handle onto the stack. It never had to go into the stack to find the handle. It is this aspect of handle pruning that makes a stack a particularly convenient data structure for implementing a shift-reduce parser.

Conflicts

. The general shift-reduce technique is:

- if there is no handle on the stack then shift
- If there is a handle then reduce

. However, what happens when there is a choice

- What action to take in case both shift and reduce are valid?

shift-reduce conflict

- Which rule to use for reduction if reduction is possible by more than one rule?

reduce-reduce conflict

. Conflicts come either because of ambiguous grammars or parsing method is not powerful enough

Shift reduce conflict

Consider the grammar E →E+E | E*E | id and
input id+id*id

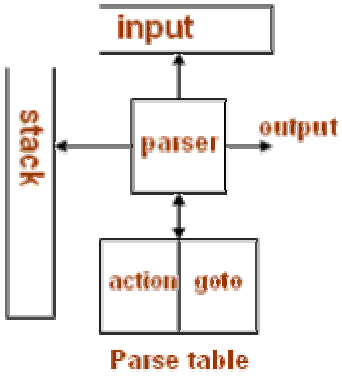
stack	**input**	**action**	**stack**	**input**	**action**
E+E	*id	reduce by E →E+E	E+E	*id	shift
E	*id	shift	E+E*	id	shift
E*	id	shift	E+E*id		reduce by E →id
E*id		reduce by E →id	E+E*E		reduce by E →E*E
E*E		reduce byE →E*E	E+E		reduce by E →E+E
E			E		

Reduce reduce conflict

Consider the grammar M →R+R | R+c | R
R →c
and input c+c

Stack	**input**	**action**	**Stack**	**input**	**action**
	c+c	shift		c+c	shift
c	+c	reduce by R →c	c	+c	reduce by R →c
R	+c	shift	R	+c	shift
R+	c	shift	R+	c	shift
R+c		reduce by R →c	R+c		reduce by M →R+c
R+R		reduce by M →R+R	M		

LR parsing



. Input contains the input string.
. Stack contains a string of the form $S_0 X_1 S_1 X_2 ..X_n S_n$ where each X_i is a grammar symbol and each S i is a state.
. Tables contain action and goto parts.
. action table is indexed by state and terminal symbols.
. goto table is indexed by state and non terminal symbols.

Actions in an LR (shift reduce) parser

. Assume S_i is top of stack and a_i is current input symbol
. Action $[S_i, a_i]$ can have four values
1. shift a_i to the stack and goto state S_j
2. reduce by a rule
3. Accept
4. error

Configurations in LR parser

Stack: $S_0 X_1 S_1 X_2 .X_m S_m$ **Input**: $a_i a_{i+1} .a_n \$$
. If action$[S_m, a_i]$ = shift S Then the configuration becomes
Stack : $S_0 X_1 S_1 ..X_m S_m a_i S$ **Input** : $a_{i+1} .a_n \$$
. If action$[S_m, a_i]$ = reduce A ß Then the configuration becomes
Stack : $S_0 X_1 S_1 .X_{m-r} S_{m-r} AS$ **Input** : $a_i a_{i+1} .a_n \$$
Where r = | ß | and S = goto$[S_{m-r}, A]$
. If action$[S_m, a_i]$ = accept
Then parsing is completed. HALT
. If action$[S_m, a_i]$ = error
Then invoke error recovery routine.

LR parsing Algorithm

Initial state: Stack: S_0 Input: w$

```
Loop{
   if action[S,a] = shift S'
      then push(a); push(S'); ip++
      else if action[S,a] = reduce A→β
              then pop (2*|β|)  symbols;
                   push(A); push (goto[S'',A])
                   (S'' is the state after popping symbols)
              else if action[S,a] = accept
                      then exit
                      else error
    }
```

Example

Consider the grammar And its parse table

E →E + T | T
T →T * F | F
F → (E) | id

State	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			

Parse id + id * id

Stack	Input	Action
0	id+id*id$	shift 5
0 id 5	+id*id$	reduce by F→id
0 F 3	+id*id$	reduce by T→F
0 T 2	+id*id$	reduce by E→T
0 E 1	+id*id$	shift 6
0 E 1 +6	id*id$	shift 5
0 E 1 +6 id 5	*id$	reduce by F→id
0 E 1 +6 F 3	*id$	reduce by T→F
0 E 1 +6 T 9	*id$	shift 7
0 E 1 +6 T 9 * 7	id$	shift 5
0 E 1 +6 T 9 * 7 id 5	$	reduce by F→id
0 E 1 +6 T 9 * 7 F 10	$	reduce by T→T*F
0 E 1 +6 T 9	$	reduce by E→E+T
0 E 1	$	ACCEPT

Parser states

. Goal is to know the valid reductions at any given point
. Summarize all possible stack prefixes a as a parser state
. Parser state is defined by a DFA state that reads in the stack α
. Accept states of DFA are unique reductions

The parser states help us in identifying valid reductions at any particular step of parsing. In a nutshell, these parser states are all the possible stack prefixes possible. During the course of parsing, the input is

consumed and we switch from one parser state to another and finally the input is accepted on unique reductions(or to say that the Deterministic Finite Automata reaches accepting state).

Constructing parse table

Augment the grammar

. G is a grammar with start symbol S

. The augmented grammar G' for G has a new start symbol S' and an additional production S' S

. When the parser reduces by this rule it will stop with accept

The first step in the process of constructing the parse table, is to augment the grammar to include one more rule. The input is accepted by the parser, only when it reduces with this rule. We add a new start symbol in the augmented grammar (say S') and another rule (S' S), where S is the start symbol in the original grammar. The parser stops and accepts only when it reduces with this rule.

Viable prefixes

.α is a viable prefix of the grammar if

- There is a w such that α w is a right sentential form
- α .w is a configuration of the shift reduce parser

. As long as the parser has viable prefixes on the stack no parser error has been seen

. The set of viable prefixes is a regular language (not obvious)

. Construct an automaton that accepts viable prefixes

The set of prefixes of right sequential forms that can appear on the stack of a shift-reduce parser are called viable prefixes. It is always possible to add terminal symbols to the end of a viable prefix to obtain a right-sentential form. Thus, as long as we are able to reduce the portion on the stack to a viable prefix, we are safe and no error occurs. Or to say, the input is accepted by the parser if we are able to manage viable prefixes as we consume the input, and at the last, we are left with the start symbol on the stack. The set of viable prefixes is a regular expression. This is not so obvious, but soon we are going to define the states for such an automaton which can accept viable prefixes.

LR(0) items

. An LR(0) item of a grammar G is a production of G with a special symbol "." at some position of the right side

. Thus production A →XYZ gives four LR(0) items

A →.XYZ

A →X.YZ

A →XY.Z

A →XYZ.

. An item indicates how much of a production has been seen at a point in the process of parsing

- Symbols on the left of "." are already on the stacks
- Symbols on the right of "." are expected in the input

An LR(0) item represents a production in such a way, that you keep track of the input already read (i.e., present on the stack) and the input yet to be expected. Consider a typical example A Y.XZ , here the special symbol " . " means that the expression to the left of it (i.e., Y) is present on the stack, while the one of its right is yet expected to complete this production.

Start state

. Start state of DFA is an empty stack corresponding to S' →.S item

- This means no input has been seen
- The parser expects to see a string derived from S

. **Closure** of a state adds items for all productions whose LHS occurs in an item in the state, just after "."

- Set of possible productions to be reduced next
- Added items have "." located at the beginning
- No symbol of these items is on the stack as yet

The start state of the parser corresponds to an empty stack corresponding the production S' .S (the production added during the augmentation). This means, that the parser expects to see input which can be reduced to S and then further to S' (in which case, the input is accepted by the parser). Closure is the method to find the set of all productions that can be reduced next. The items added by closure have " . " at the beginning of the RHS of production. To put in simple words, it adds alternate ways to expect further input.

Closure operation

. If I is a set of items for a grammar G then closure(I) is a set constructed as follows:
- Every item in I is in closure (I)
- If A →α .B ß is in closure(I) and B →γ is a production then B →. γ is in closure(I)

. Intuitively A →α .B ß indicates that we might see a string derivable from B ß as input
. If input B →γ is a production then we might see a string derivable from γ at this point

As stated earlier, closure operation requires us to find all such alternate ways to expect further input. If I is a set of items for a grammar G then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to closure (I).
2. If A →α .B ß is in closure(I) and B→ γ is a production then add the item B →. γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Intuitively A→ α .B ß in closure(**I**) indicates that, at some point in the parsing process, we think we might next see a substring derivable from B ß as input. If B→γ is a production, we also expect we might see a substring derivable from γ at this point. For this reason, we also include B →. γ in closure(**I**).

Example

Consider the grammar

E' →E
E →E + T | T
T →T * F | F
F → (E) | id

If I is { E' →.E } then closure(I) is

E' →.E
E →.E + T
E →.T
T →.T * F
T →.F
F →.id
F →.(E)

Consider the example described here. Here I contains the LR(0) item E' → .E . We seek further input which can be reduced to E. Now, we will add all the productions with E on the LHS. Here, such productions are E →.E+T and E →.T. Considering these two productions, we will need to add more productions which can reduce the input to E and T respectively. Since we have already added the productions for E, we will need those for T. Here these will be T→ .T+F and T →.F. Now we will have to add productions for F, viz.

F →.id and F →.(E).

Goto operation

. Goto(I,X) , where I is a set of items and X is a grammar symbol,
- is closure of set of item A →α X. ß

- such that A →α .X ß is in I

. Intuitively if I is a set of items for some valid prefix a then goto(I,X) is set of valid items for prefix α X

. If I is { E' →E. , E →E. + T } then goto(I,+) is

E →E + .T
T →.T * F
T →.F
F →.(E)
F →.id

The second useful function is Goto(I,X) where I is a set of items and X is a grammar symbol. Goto(I,X) is defined to be the closure of the set of all items [A →a X. ß] such that [A →a .X ß] is in I. Intuitively, if I is set of items that are valid for some viable prefix a , then goto(I,X) is set of items that are valid for the viable prefix a X. Consider the following example: If I is the set of two items { E ' →E. , E →E. + T }, then goto(I,+) consists of

E →E + .T
T →.T * F
T →.F
F →.(E)
F →.id

We computed goto(I,+) by examining I for items with + immediately to the right of the dot. E' →E. is not such an item, but E →E. + T is. We moved the dot over the + to get {E → E + .T} and the took the closure of this set.

Sets of items

C : Collection of sets of LR(0) items for grammar G'

C = { closure ({ S'→.S }) }

repeat

for each set of items I in C

and each grammar symbol X

such that goto (I,X) is not empty and not in C

ADD goto(I,X) to C

until no more additions

We are now ready to give an algorithm to construct C, the canonical collection of sets of LR(0) items for an augmented grammar G'; the algorithm is as shown below:

C = { closure ({ S' →.S }) }
repeat
for each set of items I in C and each grammar symbol X
such that goto (I,X) is not empty and not in C do
ADD goto(I,X) to C
until no more sets of items can be added to C

Example

Grammar:

E ' E
E E+T | T
T T*F | F
F (E) | id

I_0 : closure(E ' .E)

E ' .E

I_2 : goto(I_0 ,T)

E T.
T T. *F

I_3 : goto(I_0 ,F)

T F.

I_4 : goto(I_0 ,( )

F (.E)

I_6 : goto(I_1 ,+)

E E + .T
T .T * F
T .F
F .(E)
F .id

I_7 : goto(I_2 ,*)

I_9 : goto(I_6 ,T)

E E + T.
T T. * F

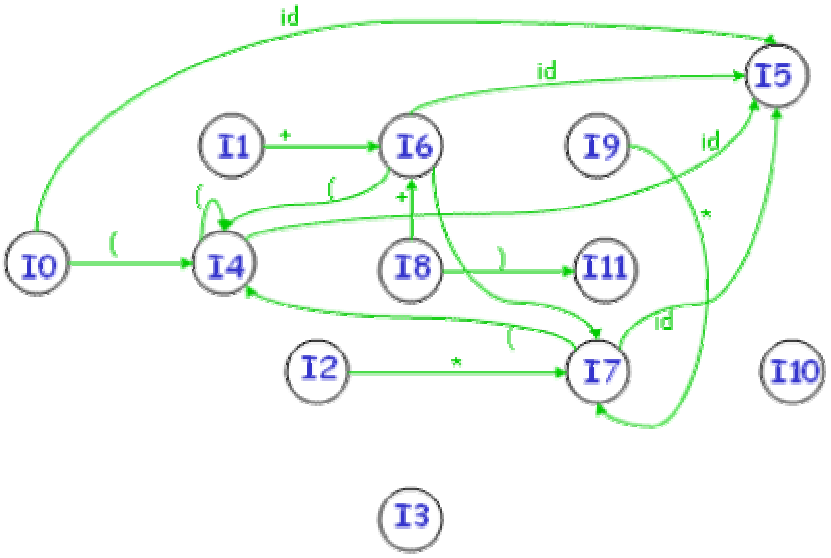
goto(I_6 ,F) is I_3
goto(I_6 ,( ) is I_4
goto(I_6 ,id) is I_5

I_{10} : goto(I_7 ,F)

E .E + T	E .E + T	T T * .F	T T * F.
E .T	E .T	F .(E)	goto(I_7 ,( ) is I_4
T .T * F	T .T * F	F .id	goto(I_7 ,id) is I_5
T .F	T .F	**I_8 : goto(I_4 ,E)**	**I_{11} : goto(I_8 ,))**
F .(E)	F .(E)	F (E.)	F (E).
F .id	F .id	E E. + T	goto(I_8 ,+) is I_6
I_1 : goto(I_0 ,E)	**I_5 : goto(I_0 ,id)**	goto(I_4 ,T) is I_2	goto(I_9 ,*) is I_7
E ' E.	F id.	goto(I_4 ,F) is I_3	
E E. + T		goto(I_4 ,( ) is I_4	
		goto(I_4 ,id) is I_5	

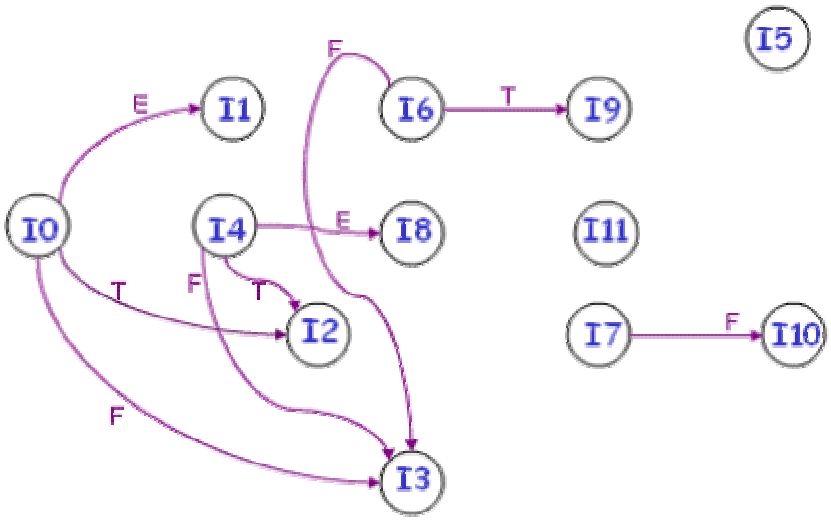
Let's take an example here. We have earlier calculated the closure I_0 . Here, notice that we need to calculate goto (I_0 ,E), goto(I_0 ,T), goto(I_0 , F) , goto (I_0 , ( ) and goto(I_0 , id). For calculating goto(I_0 , E), we take all the LR(0) items in I 0 , which expect E as input (i.e. are of the form A α .E ß), and advance ".". Closure is then taken of this set. Here, goto(I 0 , E) will be closure { E ' E. , E E.+T }. The closure adds no item to this set, and hence goto(I 0 , E)={ E ' E. , E E.+T }.

Example



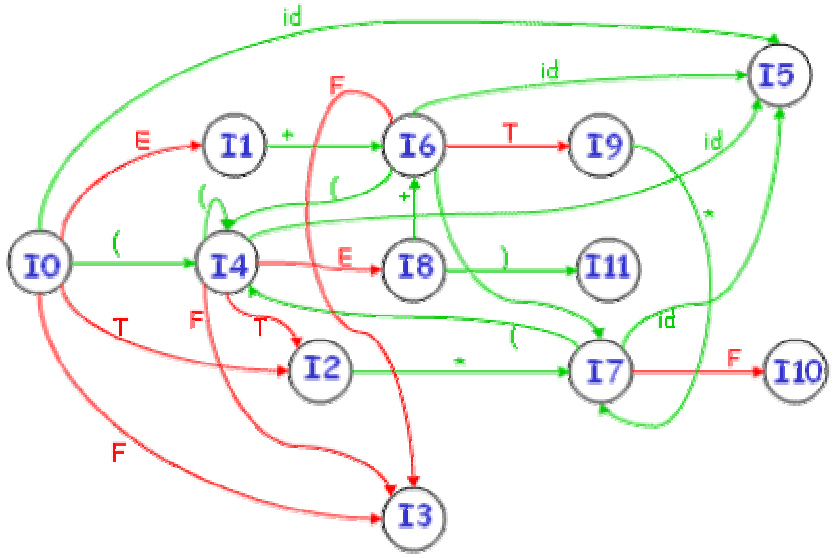
The figure shows the finite automaton containing different parser states viz. I 0 to I n and possible transitions between the states on seeing a terminal symbol i.e., from a set of LR(0) items, for every item goto(I i , x) = I j where x is a terminal, we have a transition from I i to I j on 'x'.

Example



The figure shows the finite automaton containing different parser states viz. I_0 to I_n and possible transitions between them on seeing a non- terminal symbol i.e., from set of LR(0) items, for every item goto(I_i , x) = I_j where x is a non-terminal, we have a transition from I_i to I_j on 'x'.

Example



The figure shows the finite automaton containing different parser states viz. I_0 to I_n and possible transitions between them on seeing a non- terminal symbol i.e., from set of LR(0) items, for every item goto(I_i , x) = I_j where x is a non-terminal, we have a transition from I_i to I_j on 'x'.

Construct SLR parse table

. Construct C={I_0 , . , I_n } the collection of sets of LR(0) items
. If A →α .a ß is in I_i and goto(I_i , a) = I_j then action[i,a] = shift j
. If A →a . is in I_i
then action[i,a] = reduce A →a for all a in follow(A)
. If S' →S. is in I_i then action[i,$] = accept
. If goto(I i ,A) = I_j then goto[i,A]=j for all non terminals A

. All entries not defined are errors

The SLR parse table is constructed for parser states (I_0 to I_n) against terminal and non terminal symbols . For terminals, entries are referred as 'action' for that state and terminal, while for non terminal, entries are 'goto' for state and non terminal. The way entries are filled is :

. If $A \rightarrow \alpha .a ß$ is in I_i and goto(I_i , a) = I_j where a is a terminal then action[i, a] = shift j.

. If $A \rightarrow \alpha .$ is in I_i where a is a string of terminals and non terminals then action[i, b] = reduce $A \rightarrow a$ for all b in follow(A).

. If $S' \rightarrow S.$ is in I_i where S' is symbol introduced for augmenting the grammar then action[i, $] = accept.

. If goto(I_i , A) = I_j where A is a non terminal then goto[i, A] = j.

. The entries which are not filled are errors.

Notes

. This method of parsing is called SLR (Simple LR)

. LR parsers accept LR(k) languages

- L stands for left to right scan of input
- R stands for rightmost derivation
- k stands for number of lookahead token

. SLR is the simplest of the LR parsing methods. SLR is too weak to handle most languages!

. If an SLR parse table for a grammar does not have multiple entries in any cell then the grammar is unambiguous

. All SLR grammars are unambiguous

. Are all unambiguous grammars in SLR?

This parsing method is called SLR which stands for Simple LR. LR parsers accept **LR** (**k**) languages. Here, L stands for left to right scan of input, R stands for rightmost derivation and k stands for number of look ahead tokens. SLR parsing is the simplest of all parsing methods. This method is weak and cannot handle most of the languages. For instance, SLR parsing cannot handle a language which depends not only on top of stack but the complete stack. Also, look ahead symbols cannot be handled by SLR parser.

If an SLR parse table for a grammar does not have multiple entries in any cell then the grammar is unambiguous. All SLR grammars are unambiguous but all unambiguous grammars are not in SLR.

Assignment

Construct SLR parse table for following grammar

E →E + E | E - E | E * E | E / E | (E) | digit. Show steps in parsing of string 9*5+(2+3*7)

. Steps to be followed

- Augment the grammar
- Construct set of LR(0) items
- Construct the parse table
- Show states of parser as the given string is parsed

Example

. Consider following grammar and its SLR parse table:

S ' S

S L = R
S R
L *R
L id
R L
I_0 : S' .S
S .L=R
S .R
L .*R
L .id
R .L
I_1 : goto(I_0 , S)
S' S.
I_2 : goto(I_0 , L)
S L.=R
R L.

Construct rest of the items and the parse table.

Given grammar:

S L = R
S R
L *R
L id
R L

Augmented grammar:

S ' S
S L = R
S R
L *R
L id
R L

Constructing the set of LR(0) items:

Using the rules for forming the LR(0) items, we can find parser states as follows:

I_0 : closure(S ' .S)
S ' .S
S .L = R
S .R
L .*R
L .id
R .L
I_1: goto(I_0 , S)
S ' S.
I_2 : goto(I_0 , L)
S L.=R
R L.

I_3 : goto(I_0 , R)
S R.
I_4 : goto(I_0 , *)
L *.R
R .L
L .*R
L .id
I_5 : goto(I_0 , id)
L id.
I_6 : goto(I_2 , =)
S L=.R
R .L
L .*R
L .id
I_7 : goto(I_4 , R)
L *R.
I_8 : goto(I_4 , L)
R L.
goto(I_4 , *) = I_4
goto(I_4 , id) = I_5
I_9 : goto(I_6 , R) S L=R.
I_{10} : goto(I_6 , L) R L.
goto(I_6, *) = I_4
goto(I_6 , id) = I_5
So, the set is C = { I0, I1, I2, ... I10 }

SLR parse table for the grammar

	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				acc			
2	s6,r6			r6			
3				r3			
4		s4	s5			8	7
5	r5			r5			
6		s4	s5			8	9
7	r4			r4			
8	r6			r6			
9				r2			

The table has multiple entries in action[2,=]
Using the rules given for parse table construction and above set of LR(0) items, parse table as shown can be constructed. For example:
Consider [6,*] : I_6 contains L .*R and goto (I_6 , *) = I_4. So [6,*] = shift 4 or s4.
Consider [4,L] : goto(I_4 , L) = I_8. So [4,L] = 8.
Consider [7,=] : I_7 contains L *R. and ' = ' is in follow(L).
So [7,=] = reduce L *R or r4 i.e. reduce by 4th rule.
Similarly the other entries can be filled .

Consider the entry for [2,=]
I_2 contains S L.=R and goto(I_2 , =) = I_6. So [2,=] contains 'shift 6'.
I_2 contains R L. and '=' is in follow(R).
So [2,=] contains 'reduce 6'. So [2,=] has multiple entries viz. r6 and s6.

There is both a shift and a reduce entry in action[2,=]. Therefore state 2 has a shift-reduce conflict on symbol "=", However, the grammar is not ambiguous.

. Parse id=id assuming reduce action is taken in [2,=]

Stack	**input**	**action**
0	id=id	shift 5
0	=id	reduce by L id
0 L 2	=id	reduce by R L
0 R 3	=id	error

. if shift action is taken in [2,=]

Stack	**input**	**action**
0	id=id$	shift 5
0 id 5	=id$	reduce by L id
0 L 2	=id$	shift 6
0 L 2 = 6	id$	shift 5
0 L 2 = 6 id 5	$	reduce by L id
0 L 2 = 6 L 8	$	reduce by R L
0 L 2 = 6 R 9	$	reduce by S L=R
0 S 1	$	ACCEPT

We can see that [2,=] has multiple entries, one shift and other reduce, which makes the given grammar ambiguous but it is not so. 'id = id' is a valid string in S' as

S ' S L=R L=L L=id id=id

but of the given two possible derivations, one of them accepts it if we use shift operation while if we use reduce at the same place, it gives error as in the other derivation.

Problems in SLR parsing

. No sentential form of this grammar can start with R=.
. However, the reduce action in action[2,=] generates a sentential form starting with R=
. Therefore, the reduce action is incorrect
. In SLR parsing method state i calls for reduction on symbol "a", by rule A α if I i contains [A α.] and "a" is in follow(A)
. However, when state I appears on the top of the stack, the viable prefix ßα on the stack may be such that ß A can not be followed by symbol "a" in any right sentential form
. Thus, the reduction by the rule A α on symbol "a" is invalid
. SLR parsers cannot remember the left context

The reason for the above is that in derivation which gave an error, if we inspect the stack while reducing for [2,=], it would generate a sentential form 'R='which is incorrect as can be seen from the productions. Hence using the reduce action is incorrect and other one is the correct derivation. So, the grammar is not allowing multiple derivations for [2,=] and thus, is unambiguous.

The reason why using this reduction is incorrect is due to limitation of SLR parsing. A reduction is done in SLR parsing on symbol b in state i using rule A a if b is in follow(A) and I i contains [A a.] but it might be possible that viable prefix on stack ß a may be such that ß A cannot be followed by 'b' in any right sentential form, hence doing the reduction is invalid.

So, in given example, in state 2 on symbol ' = ' , we can reduce using R L but the viable prefix is 0L which on reduction gives 0R, which cannot be followed by '=' in any right sentential form as can be seen from the grammar. So, we can conclude SLR parsers cannot remember left context which makes it weak to handle all languages.

Canonical LR Parsing

. Carry extra information in the state so that wrong reductions by A α will be ruled out

. Redefine LR items to include a terminal symbol as a second component (look ahead symbol)

. The general form of the item becomes [A α . ß , a] which is called LR(1) item.

. Item [A α ., a] calls for reduction only if next input is a. The set of symbols

Canonical LR parsers solve this problem by storing extra information in the state itself. The problem we have with SLR parsers is because it does reduction even for those symbols of follow(A) for which it is invalid. So LR items are redefined to store 1 terminal (look ahead symbol) along with state and thus, the items now are LR(1) items.

An LR(1) item has the form : [A a . ß , a] and reduction is done using this rule only if input is 'a'. Clearly the symbols a's form a subset of follow(A).

Closure(I)

repeat

for each item [A α .B ß , a] in I

for each production B γ in G'

and for each terminal b in First(ß a)

add item [B . γ , b] to I

until no more additions to I

To find closure for Canonical LR parsers:

Repeat

for each item [A α .B ß , a] in I

for each production B γ in G'

and for each terminal b in First(ß a)

add item [B . γ , b] to I

until no more items can be added to I

Example

Consider the following grammar

S' S

S CC

C cC | d

Compute closure(I) where I={[S' .S, $]}

S' .S, $

S .CC, $

C .cC, c

C .cC, d

C .d, c

C .d, d

For the given grammar:

S ' S

S CC

C cC | d

I : closure([S ' S, $])

S '	**.S**	$	as first(e $) = {$}
S	**.CC**	$	as first(C$) = first(C) = {c, d}
C	**.cC**	c	as first(Cc) = first(C) = {c, d}
C	**.cC**	d	as first(Cd) = first(C) = {c, d}
C	**.d**	c	as first(e c) = {c}
C	**.d**	d	as first(e d) = {d}

Example

Construct sets of LR(1) items for the grammar on previous slide

I_0 :	S '	.S,	$
	S	.CC,	$
	C	.cC,	c /d
	C	.d,	c /d
I_1 :	goto(I_0 ,S)		
	S '	S.,	$
I_2 :	goto(I 0 ,C)		
	S	C.C,	$
	C	.cC,	$
	C	.d,	$
I_3 :	goto(I 0 ,c)		
	C	c.C,	c/d
	C	.cC,	c/d
	C	.d,	c/d
I_4 :	goto(I 0 ,d)		
	C	d.,	c/d
I_5 :	goto(I 2 ,C)		
	S	CC.,	$
I_6 :	goto(I 2 ,c)		
	C	c.C,	$
	C	.cC,	$
	C	.d,	$
I_7 :	goto(I 2 ,d)		
	C	d.,	$
I_8 :	goto(I 3 ,C)		
	C	cC.,	c/d
I_9 :	goto(I 6 ,C)		
	C	cC.,	$

To construct sets of LR(1) items for the grammar given in previous slide we will begin by computing closure of {[S ´ .S, $]}.

To compute closure we use the function given previously.

In this case α = ε , B = S, ß =ε and a=$. So add item [S .CC, $].

Now first(C$) contains c and d so we add following items
we have A=S, α = ε , B = C, ß=C and a=$
Now first(C$) = first(C) contains c and d
so we add the items [C .cC, c], [C .cC, d], [C .dC, c], [C .dC, d].
Similarly we use this function and construct all sets of LR(1) items.

Construction of Canonical LR parse table

. Construct C={I_0 , . , I_n } the sets of LR(1) items.
. If [A α .a ß , b] is in I_i and goto(I_i , a)=I_j then action[i,a]=shift j
. If [A α ., a] is in I_i then action[i,a] reduce A α
. If [S ' S., $] is in I_i then action[i,$] = accept
. If goto(I i , A) = I_j then goto[i,A] = j for all non

We are representing shift j as sj and reduction by rule number j as rj. Note that entries corresponding to [state, terminal] are related to action table and [state, non-terminal] related to goto table. We have [1,$] as accept because [S ´ S., $] ε I_1 .

Parse table

State	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		

We are representing shift j as sj and reduction by rule number j as rj. Note that entries corresponding to [state, terminal] are related to action table and [state, non-terminal] related to goto table. We have [1,$] as accept because [S ´ S., $] ε I_1

Notes on Canonical LR Parser

. Consider the grammar discussed in the previous two slides. The language specified by the grammar is c*dc*d.
. When reading input cc.dcc.d the parser shifts cs into stack and then goes into state 4 after reading d. It then calls for reduction by C d if following symbol is c or d.
. IF $ follows the first d then input string is c*d which is not in the language; parser declares an error
. On an error canonical LR parser never makes a wrong shift/reduce move. It immediately declares an error
. **Problem** : Canonical LR parse table has a large number of states

An LR parser will not make any wrong shift/reduce unless there is an error. But the number of states in LR parse table is too large. To reduce number of states we will combine all states which have same core and different look ahead symbol.

LALR Parse table

. Look Ahead LR parsers

. Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items

I_4 : C d. , c/d I_7 : C d., $

. Replace I_4 and I_7 by a new state I_{47} consisting of (C d., c/d/$)

. Similarly I_3 & I_6 and I_8 & I_9 form pairs

. Merge LR(1) items having the same core

We will combine Ii and Ij to construct new Iij if Ii and Ij have the same core and the difference is only in look ahead symbols. After merging the sets of LR(1) items for previous example will be as follows:

I_0 : S' S $
S .CC $
C .cC c/d
C .d c/d

I_1 : goto(I 0 ,S)
S' S. $

I_2 : goto(I 0 ,C)
S C.C $
C .cC $
C .d $

I_{36} : goto(I 2 ,c)
C c.C c/d/$
C .cC c/d/$
C .d c/d/$

I_4 : goto(I 0 ,d)
C d. c/d

I_5 : goto(I 2 ,C)
S CC. $

I_7 : goto(I 2 ,d)
C d. $

I_{89} : goto(I 36 ,C)
C cC. c/d/$

Construct LALR parse table

. Construct C={I_0 , .. ,I_n } set of LR(1) items

. For each core present in LR(1) items find all sets having the same core and replace these sets by their union

. Let C' = {J_0 , .. .,J_m } be the resulting set of items

. Construct action table as was done earlier

. Let J = I_1 U I_2 .. .U I_k

since I_1 , I_2 .. ., I_k have same core, goto(J,X) will have he same core

Let K=goto(I_1 ,X) U goto(I_2 ,X) .. goto(I_k ,X) the goto(J,X)=K

The construction rules for LALR parse table are similar to construction of LR(1) parse table.

LALR parse table .

State	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		

The construction rules for LALR parse table are similar to construction of LR(1) parse table

Notes on LALR parse table

. Modified parser behaves as original except that it will reduce C d on inputs like ccd. The error will eventually be caught before any more symbols are shifted.

. In general core is a set of LR(0) items and LR(1) grammar may produce more than one set of items with the same core.

. Merging items never produces shift/reduce conflicts but may produce reduce/reduce conflicts.

. SLR and LALR parse tables have same number of states.

Canonical parser will catch error as soon as it is encountered, since it manages a state for each possible look-ahead, but on the other hand LALR parser does not distinguish between the possible lookaheads. Consider the example, where the grammar accepts c*dc*d. Suppose the parser is given input ccd. While canonical parser will exit with error status as soon as it sees d, because it does not expect $ to be seen after first d. On the other hand, LALR parser will first reduce C d, but will eventually report an error. It does not produce shift/reduce conflict since it keeps track of possible look-ahead symbol (not whole follow set) and hence is able to find when a reduction is possible. But since we are not managing each of the different look-ahead, reduce/reduce conflicts can not be avoided

Notes on LALR parse table.

. Merging items may result into conflicts in LALR parsers which did not exist in LR parsers

. New conflicts can not be of shift reduce kind:

- Assume there is a shift reduce conflict in some state of LALR parser with items {[X α .,a],[Y γ .a ß ,b]}
- Then there must have been a state in the LR parser with the same core
- Contradiction; because LR parser did not have conflicts

. LALR parser can have new reduce-reduce conflicts

- Assume states {[X α ., a], [Y ß ., b]} and {[X α ., b], [Y ß ., a]}
- Merging the two states produces {[X α ., a/b], [Y ß ., a/b]}

Merging states will never give rise to shift-reduce conflicts but may give reduce-reduce conflicts and have some grammars which were in canonical LR parser may become ambiguous in LALR parser. To realize this, suppose in the union there is a conflict on lookahead a because there is an item [A →α . ,a] calling for a reduction by A →α , and there is another item [B →ß .a γ ,b] calling for a shift. Then some set of items from which the union was formed has item [A →α . ,a], and since the cores of all these states are the same, it must have an item [B→ß .a γ ,c] for some c. But then this state has the same shift/reduce conflict on a, and the grammar was not LR(1) as we assumed. Thus the merging of states with common core can never produce a shift/reduce conflict that was not present in one of the original states, because shift actions depend only on the core, not the lookahead. But LALR parser can have

reduce-reduce conflicts. Assume states {[X →α ., a], [Y→ß ., b]} and {[X →α ., b], [Y→ß ., a]}. Now, merging the two states produces {[X→α ., a/b], [Y →ß ., a/b]} which generates a reduce-reduce conflict, since reductions by both X →α and Y →ß are called for on inputs a and b.

Notes on LALR parse table.

. LALR parsers are not built by first making canonical LR parse tables
. There are direct, complicated but efficient algorithms to develop LALR parsers
. Relative power of various classes
- SLR(1) ≤ LALR(1) ≤ LR(1)
- SLR(k) ≤ LALR(k) ≤ LR(k)
- LL(k)≤ LR(k)

In practice, we never first construct a LR parse table and then convert it into a LALR parser but there are different efficient algorithms which straight away give us the LALR parser. In fact, YACC and BISON generate LALR parser using these algorithms. Relative power of various classes:

S**LR (1) <LALR (1) <LR (1)**

SLR (K) <LALR (1) <LR (K)
LL (K) < LR (K)

But LL (K1) < LR (K2) IF K1>K2 for some large enough value of K1.

Error Recovery

. An error is detected when an entry in the action table is found to be empty.
. Panic mode error recovery can be implemented as follows:
- scan down the stack until a state S with a goto on a particular nonterminal A is found.
- discard zero or more input symbols until a symbol a is found that can legitimately follow A.
- stack the state goto[S,A] and resume parsing.

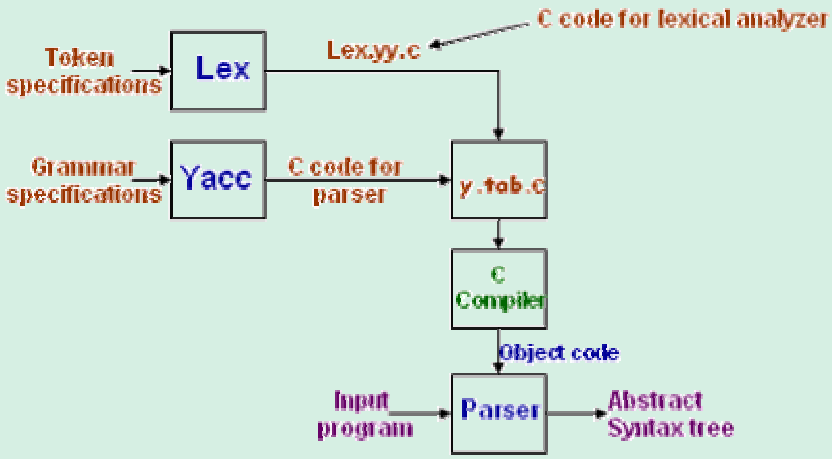
. **Choice of A:** Normally these are non terminals representing major program pieces such as an expression, statement or a block. For example if A is the nonterminal stmt, a might be semicolon or end.

Parser Generator

. Some common parser generators
- YACC: **Y** et **A** nother **C** ompiler **C** ompiler
- Bison: GNU Software
- ANTLR: **AN** other **T** ool for **L** anguage **R** ecognition

. Yacc/Bison source program specification (accept LALR grammars)
declaration
%%
translation rules
%%
supporting C routines

Yacc and Lex schema



Refer to YACC Manual