Parsing techniques

Introduction

- Context free grammars can be used to define the syntax of a programming language
- After lexical analysis, the input to a parser is typically a sequence of tokens
- The output can be of many different forms
  - A representation of the parse tree

Parsing techniques

- Shift Reduce
- Operator precedence
  - Suitable for parsing expressions
  - Precedence and associativity guide the parse
  - recognizing inputs that are not in the language of the underling grammar
- Recursive descent
  - Used for the rest of the language when operator precedence is used for parsing expression
  - Uses a collection of mutually recursive routines
  - When augmented with backtracking can produce unexpected result
- Newer methods
  - LL parsing
  - LR parsing

parsers

- A parser of as grammar is a program that takes as input a string w and produces as output either
  - a parse tree for w, if w is a sentence
  - or an error message indicating that w is not a sentence of G
- Two basic types of parsing methods for context free grammars
- Bottom up
- Top down

Bottom up Parsing

- Bottom up parsers build parse trees from the bottom (leaves) to the top of the tree (root)
- Top down parsers build parse trees starting from the root and work down to the leaves.

Representation of a parse tree

- There are two basic types of representations of a parse tree
  - Implicit
    - The sequence of productions used in some derivation
  - Explicit
    - A linked list structure for the parse tree
- Leftmost derivations
  - If we construct a parse tree in preorder, then the order in which the nodes are created corresponds to the order in which the productions are applied in leftmost derivations
- Rightmost derivations
  - Also called as canonical derivations

CFG for valid declaration of identifiers in C

- G=({dec_stat,dattype, identifier,ident_list},{int,float,double,char,id,,,;},P,dec_stat}
- P={

  dec_stat→dattype identifier ident_list

  dattype →int|float|double|char

  ident_list →, identifer ident_list | ;

  identifier →id

  }

derivation

- Construct a leftmost derivation of the string

int id,id;

dec_stat ⇒ dattype identifier ident_list

⇒ int identifier ident_list

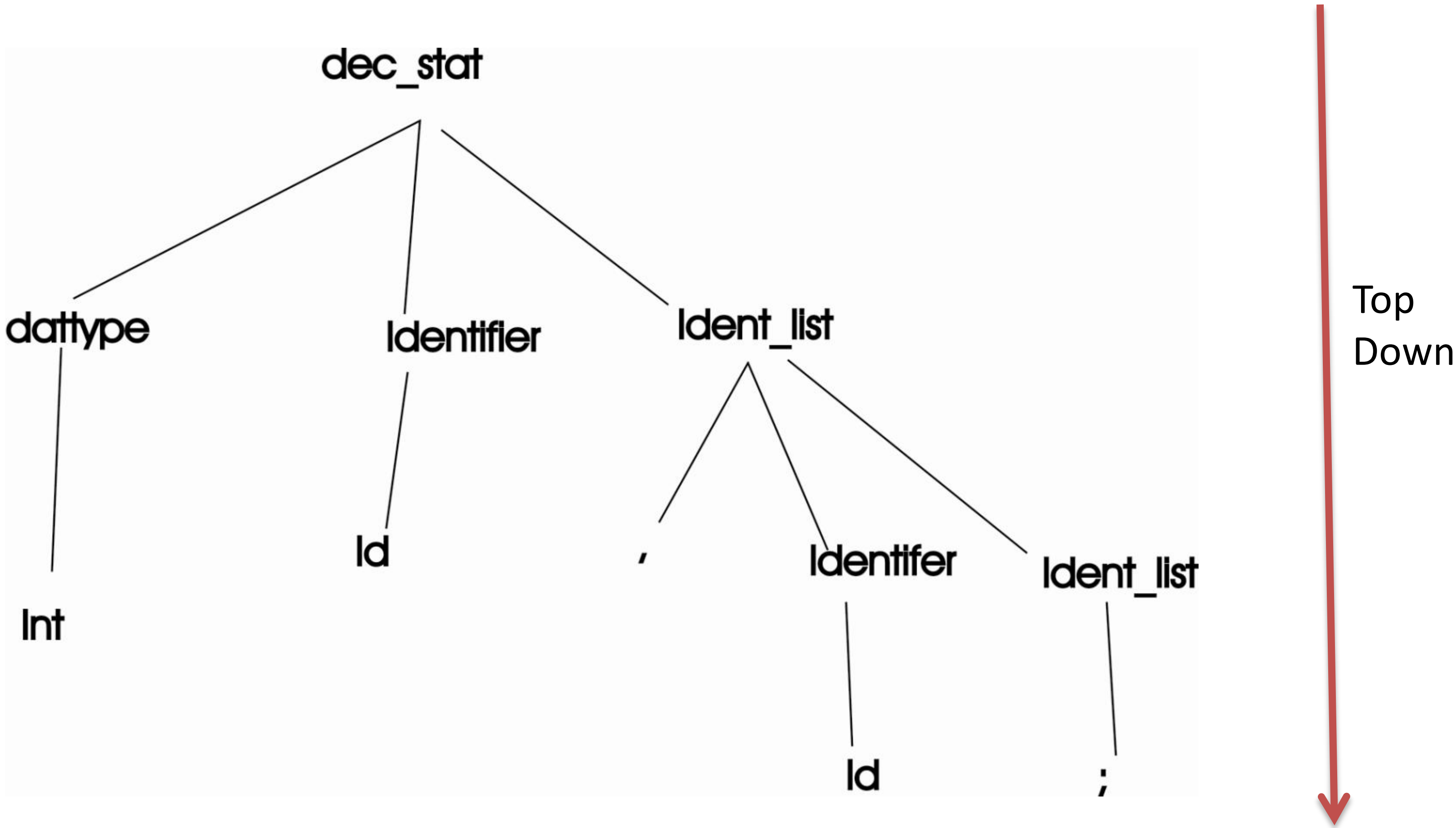
⇒ int id ident_list

⇒ int id , identifier ident_list

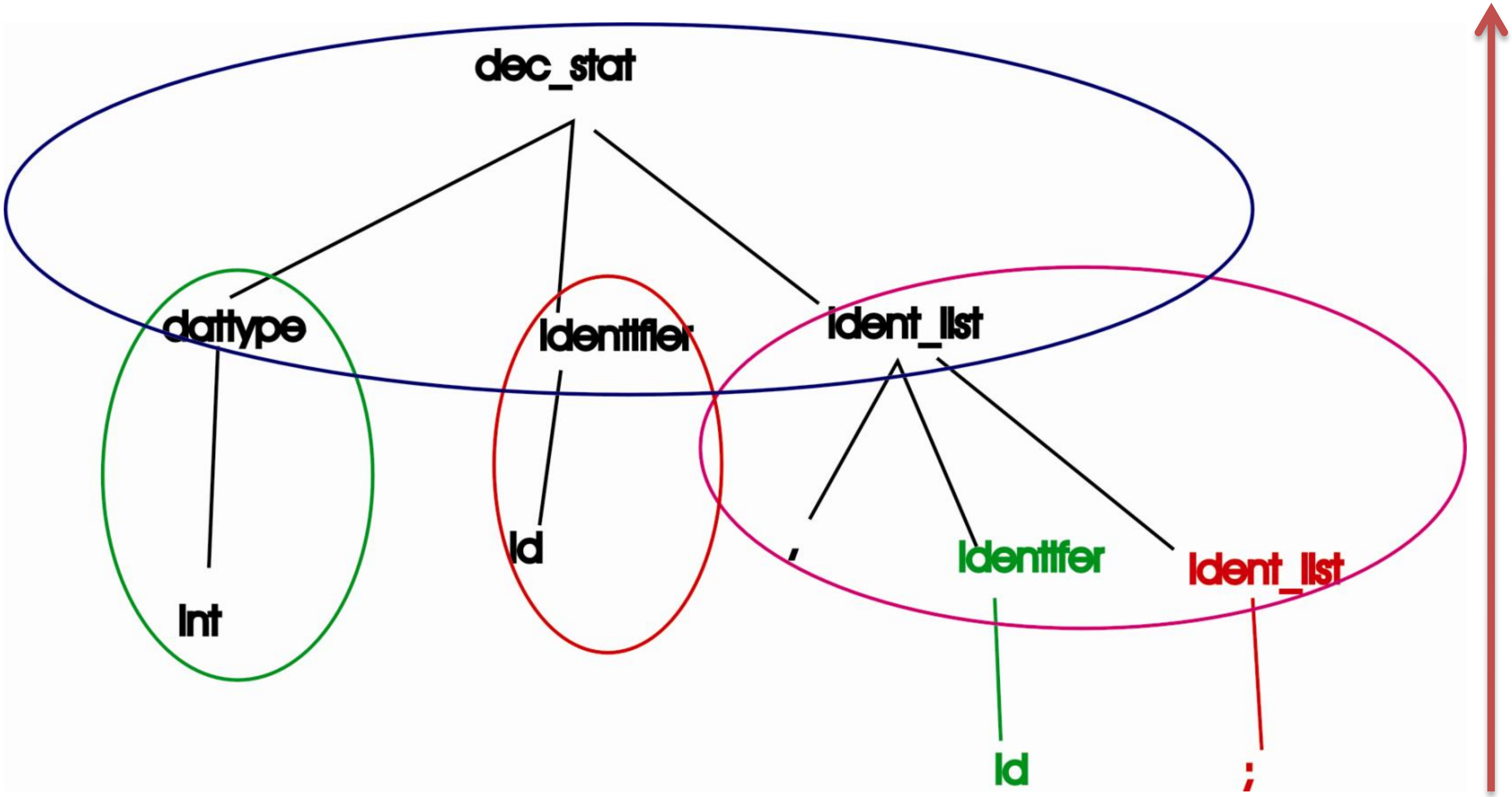
⇒ int id, id ident_list

⇒ int id, id ;

Building a parse tree top down



Building a parse tree bottom up



Bottom
Up

Shift Reduce Parsing

- Shift reduce parsing is a bottom up style of parsing
- It build a parse tree for an input string beginning at the leaves and working upwards to the root
  - Analogous to reducing a string *w* to the start symbol of a grammar

Shift Reduce cont…

- Consider the grammar
  - $S \rightarrow aAcBe$
  - $A \rightarrow Ab \mid b$
  - $B \rightarrow d$
- The string *w* is
  - *abbcde*
- Reduce the string *w* to S

Handle

- reduction - Each replacement of the right side of the production by the left side is called a *reduction*
  - These reductions in fact traced out a right most derivation in reverse
- handle - a substring which is the right side of the production such that replacement of that substring by the production left side leads eventually to a reduction to the start symbol, by the reverse of a rightmost derivation is called a “handle”
- The process of bottom up parsing may be viewed as one of finding and reducing handles

Handle

- A *handle* of a right-sentential form γ is a **production** $A \rightarrow \beta$ and a **position** of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

- i.e if $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A w \Rightarrow \alpha \beta w$, then $A \rightarrow \beta$ in the position

  following α is a handle of $\alpha \beta w$. The string w to the right of the handle contains only terminal symbols.
- If a grammar is *unambiguous* , then every right sentential form of the grammar has exactly **one handle**

Handle pruning

- A rightmost derivation in reverse is called a canonical reduction sequence which is obtained by "handle pruning"
  - start with a string of terminals *w*
  - if *w* is a string of the grammar at hand then
  - $w=\gamma_n$, where γ_n is the nth right sentential form of some as yet unknown rightmost derivation
    - $S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$
  - to construct the above derivation in reverse order we locate the handle β_n in γ_n and replace β_n by the left side of the production $A_n \rightarrow \beta_n$ to obtain the (n-1) right sentential form γ_{n-1}
- Repeat this process
- if we get S then success else failure

- Consider the grammar

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

W=id + id * id

Rightmost derivation

$E \Rightarrow E+E$

$\Rightarrow E+E*E$

$\Rightarrow E+E*id$

$\Rightarrow E+id*id$

$\Rightarrow id+id*id$

Example

- Reduction of the string w to E
- w=id + id * id

Right sentential form	**handle**	**reducing production**
Id + id * id	id	E →id
E+id * id	id	E →id
E+E*id	id	E →id
E+E*E	E * E	E →E * E
E+E	E+E	E→E + E
E		

Stack implementation of Shift-Reduce parsing

- There are two problems if we are to automate parsing by handle pruning
  1. how to locate a handle in a right sentential form
  2. what production to choose in case there is more than one production with the same right side
- Data structures used in a handle-pruning parser
  - stack and an input buffer
  - $ marks the bottom of the stack and the right end of the input

Stack implementation of Shift-Reduce parsing

- The parser operates by shifting zero or more symbols onto the stack
- The parser then reduces to the left side of the appropriate production
- The parser repeat this cycle until either
  - It detect an error
  - The stack contains the start symbol and input is empty

Stack	input
$	w$
…	
$S	$ Parser halts after successful completion

Example

- Consider id+id*id
- Shift-reduce parsing

Stack	Input	Action
$	Id+id*id$	Shift
$id	+id*id$	Reduce by E→id
$E	+id*id$	Shift
$E+	Id*id$	Shift
$E+id	*id$	Reduce by E→id
$E+E	*id$	Shift
$E+E*	Id$	Shift
$E+E*id	$	Reduce by E→id
$E+E*E	$	Reduce E→E*E
$E+E	$	Reduce E→E+E
$E	$	accept

Primary operations of the Parser

- Shift-reduce parser has four possible actions to make- *shift, reduce, accept, error*

1. **Shift**- the next input symbol is shifted to the top of the stack
2. **Reduce-**the handle is at the top of the stack and decide what nonterminal to replace the handle
3. **Accept**-announces successful completion of parsing
4. **Error**- parser discovers that a syntax error has occurred and calls an error recovery routine

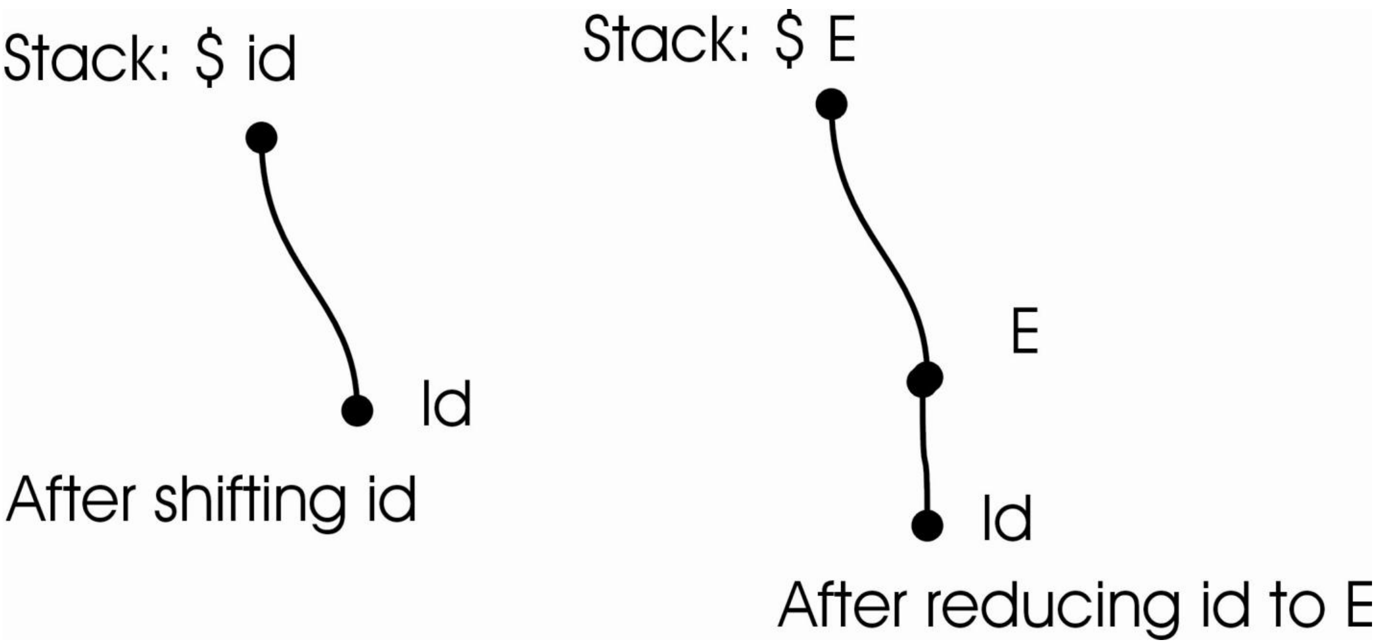
Construction of a Parse tree

- Strategy - maintain a **forest** of partially-completed derivation trees as we parse bottom up
- With **each** symbol on the stack we associate a pointer to a tree whose *root* is that symbol and whose *yield* is the string of terminals which have been reduced to that symbol
- At the end of the shift-reduce parse, the start symbol remaining on the stack will have the entire parse tree associated with it

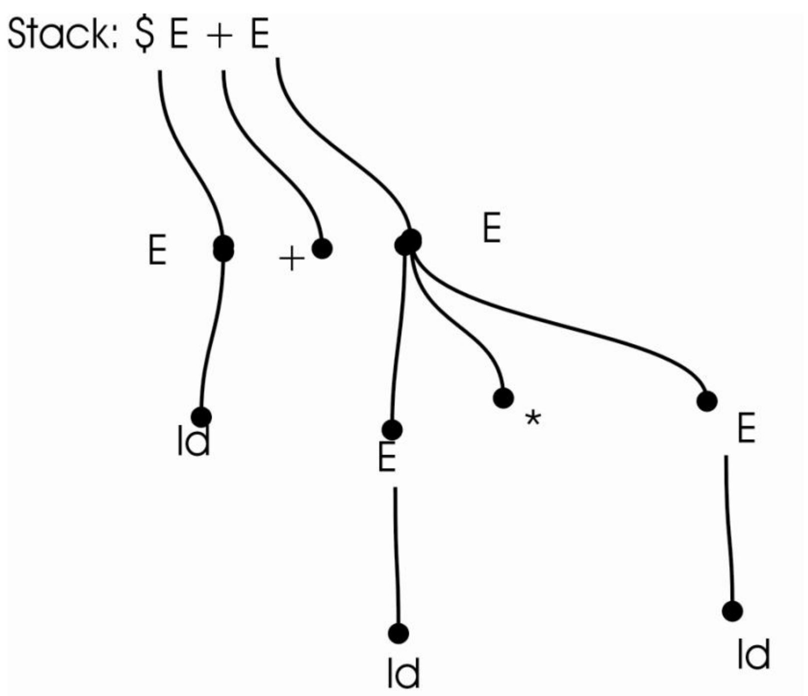
Construction of a Parse tree

1. when we shift an input symbol *a* onto the stack we create a one-node tree labeled *a* (both root and yield is *a*)
2. when we reduce $X_1, X_2, \ldots X_n$ to A, we create a new node labeled *A.* Its children are $X_1, X_2, \ldots X_n$
3. If for all *i* the tree X_i has yield x_i then the yield for the new tree is $x_1, x_2, \ldots x_n$.
4. If we reduce $\in$ to A, we create a node labeled A with one child labeled $\in$

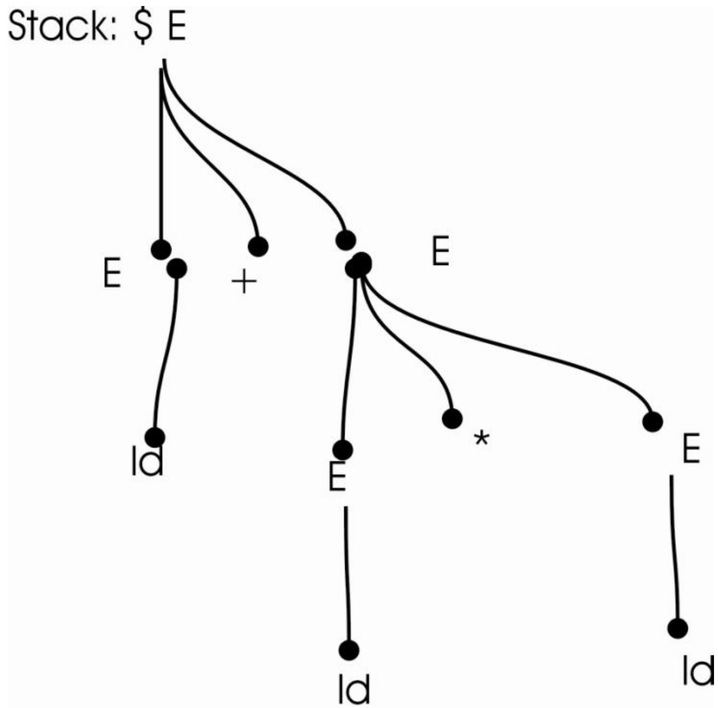
Parse tree construction



Parse tree construction



After reducing id + id * id to E+E



At completion