CS V sem Compiler Construction

Solution

Q1 (a) Why do we need syntax tree when constructing compilers ?

(b) What are the fundamental differences between parse trees and abstract syntax trees ?

Solu

(a)

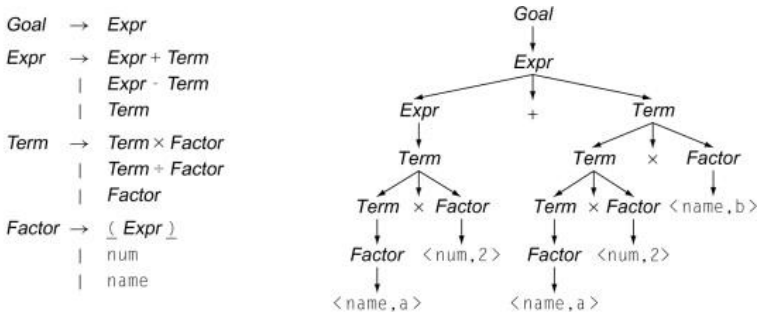
Need of Syntax tree in compilers

- Syntax tree is a way of representing the syntax of a programming language as a hierarchical tree-like structure.
- It is used for generating symbol tables for compilers and later code generation.
- Syntax tree represents all of the constructs in the language and their subsequent rules.
- It helps to preserve variable types , as well as the location of each declaration in source code.
- The order of executable statements must be explicitly represented and well defined.
- Left and right components of binary operations must be stored and correctly identified.
- Identifiers and their assigned values must be stored for assignment statements.

(b)

What is Parse Tree

A parse tree represents the syntactic structure of a string according to some context-free grammar. It describes the syntax of the input language. A parse tree does not use distinct symbol shapes for different types of constituents. The basis to construct a parse tree is phrase structure grammars or dependency grammars. It is possible to generate parse trees for natural language sentences and when processing programming languages.



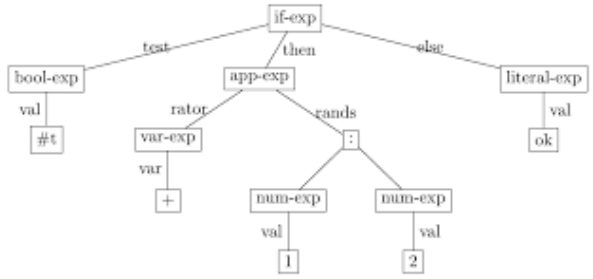
(a) Classic Expression Grammar

(b) Parse Tree for a×2+a×2×b

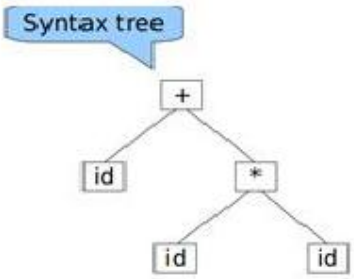
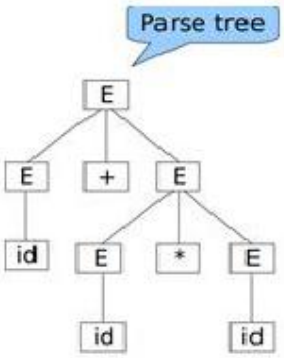
Moreover, a phrase marker is a linguistic expression marked as to its phrase structure. A tree or a bracketed expression represents it. Applying phrase structure rules to parse tree generates phrase markers. A set of possible parse trees for a syntactically ambiguous sentence is a parse forest.

What is Syntax Tree

A syntax tree describes the abstract syntactic structure of source code written in a programming language. It focuses on the rules rather than elements such as braces, semicolons that terminate statements in some languages. Also, it is a hierarchy with the elements of programming statements divided into several sections. The nodes of the tree signify a construct occurring in the source code. It does not represent every detail in the real syntax; instead, it shows the structural based and content-based details. Subsequent processing such as contextual analysis adds extra information to the syntax tree.



Syntax tree helps to determine the accuracy of the compiler. If the syntax tree contains an error, the compiler displays an error message. Program analysis and program transformation are some other uses of the syntax tree.

Syntax Tree	Parse Tree
interior nodes are "operators", leaves are operands	interior nodes are non-terminals, leaves are terminals
when representing a program in a tree structure usually use a syntax tree	rarely constructed as a data structure
Represents the abstract syntax of a program (the semantics)	Represents the concrete syntax of a program
Contain only meaningful information.	Contain unusable information also.
Syntax tree examples Grammar: E → E * E | E + E | id Program: a + b * c 	Parse tree examples Grammar: E → E * E | E + E | id Program: a + b * c 

Q 2 Write short notes on following

a. operator precedence parser for regular expression

b. difference between bottom up and top down parsing with example

c. YACC error handling in LR parser.

d. Context Free Grammar.

Solu

(a) Operator precedence parser –

An operator precedence parser is a one of the bottom-up parser that interprets an operator-precedence grammar. This parser is only used for operator grammars. *Ambiguous grammars are not allowed* in case of any parser except operator precedence parser.

There are two methods for determining what precedence relations should hold between a pair of terminals:

1. Use the conventional associativity and precedence of operator.
2. The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees.

This parser relies on the following three precedence relations: **⋖, ≐, ⋗**

a ⋖ b This means a "yields precedence to" b.

a ⋗ b This means a "takes precedence over" b.

a ≐ b This means a "has precedence as" b.

	id	+	*	$
id		⋗	⋗	⋗
+	⋖	⋗	⋖	⋗
*	⋖	⋗	⋗	⋗
$	⋖	⋖	⋖	

Figure – Operator precedence relation table for grammar E->E+E/E*E/id

There is not given any relation between id and id as id will not be compared and two variables can not come side by side. There is also a disadvantage of this table as if we have n operators than size of table will be n*n and complexity will be $0(n^2)$. In order to increase the size of table, use **operator function table**.

The operator precedence parsers usually do not store the precedence table with the relations; rather they are implemented in a special way. Operator precedence parsers use **precedence functions** that map terminal symbols to integers, and so the precedence relations between the symbols are implemented by numerical comparison. The parsing table can be encoded by two precedence functions **f** and **g** that map terminal symbols to integers. We select f and g such that:

1. f(a) < g(b) whenever a is precedence to b
2. f(a) = g(b) whenever a and b having precedence
3. f(a) > g(b) whenever a takes precedence over b

(b) Top-down vs Bottom-up Parsing

BASIS FOR COMPARISON	TOP-DOWN PARSING	BOTTOM-UP PARSING
Initiates from	Root	Leaves
Working	Production is used to derive and check the similarity in the string.	Starts from the token and then go to the start symbol.

Uses	Backtracking (sometimes)	Handling
Strength	Moderate	More powerful
Producing a parser	Simple	Hard
Type of derivation	Leftmost derivation	Rightmost derivation

(d) Context-Free Grammars

A context-free grammar (CFG) is a set of recursive rewriting rules (or productions) used to generate patterns of strings.

A CFG consists of the following components:

- a set of *terminal symbols*, which are the characters of the alphabet that appear in the strings generated by the grammar.
- a set of *nonterminal symbols*, which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
- a set of *productions*, which are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).
- a *start symbol*, which is a special nonterminal symbol that appears in the initial string generated by the grammar.

To generate a string of terminal symbols from a CFG, we:

- Begin with a string consisting of the start symbol;
- Apply one of the productions with the start symbol on the left hand size, replacing the start symbol with the right hand side of the production;
- Repeat the process of selecting nonterminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all nonterminals have been replaced by terminal symbols.

Classification of Context Free Grammars

Context **F**ree **G**rammars (CFG) can be classified on the basis of following two properties:

1) Based on number of strings it generates.

- If CFG is generating finite number of strings, then CFG is **Non-Recursive** (or the grammar is said to be Non-recursive grammar)
- If CFG can generate infinite number of strings then the grammar is said to be **Recursive** grammar

During Compilation, the parser uses the grammar of the language to make a parse tree(or derivation tree) out of the source code. The grammar used must be unambiguous. An ambiguous

grammar must not be used for parsing.

2) Based on number of derivation trees.

- If there is only 1 derivation tree then the CFG is unambiguous.

If there are more than 1 derivation tree, then the CFG is ambiguous

Example:

$L= \{wcw^R \mid w € (a, b)^*\}$

Production rules:

1. $S \rightarrow aSa$
2. $S \rightarrow bSb$
3. $S \rightarrow c$

Now check that abbcbba string can be derived from the given CFG.

4. $S \Rightarrow aSa$
5. $S \Rightarrow abSba$
6. $S \Rightarrow abbSbba$
7. $S \Rightarrow abbcbba$

(C) Error Handling in Compiler Design

The tasks of the **Error Handling** process are to detect each error, report it to the user, and then make some recover strategy and implement them to handle error. During this whole process processing time of program should not be slow. An **Error** is the blank entries in the symbol table.

Types or Sources of Error – There are two types of error: run-time and compile-time error:

1. A **run-time error** is an error which takes place during the execution of a program, and usually happens because of adverse system parameters or invalid input data. The lack of sufficient memory to run an application or a memory conflict with another program and logical error are example of this. Logic errors, occur when executed code does not produce the expected result. Logic errors are best handled by meticulous program debugging.
2. **Compile-time errors** rises at compile time, before execution of the program. Syntax error or missing file reference that prevents the program from successfully compiling is the example of this.

YACC

Yacc is also capable of more sophisticated error-handling. For example, we can tell the parser to discard some of the subsequent tokens, too, simply by putting a token after the error token.

```
menu_command :   label  default  EXEC        '\n'
        |   label  default  olvwm_cmd    '\n'
        |   label  default  olvwm_cmd_arg '\n'
        |   label  error              '\n'
                        { printf("Invalid menu item (%s) at line %d is invalid\n",
                                $1,@1.first_line);
                        }
        ;
```

In this case, the parser

- Discards everything on the stack going back to the label token
- Keeps reading tokens up till the next '\n', and discards them, too.
- Uses the tokens label error and '\n' to complete the rule for menu_command.

In this case, the rule

```
label error '\n'
```

is not much different to just having

```
error '\n'
```

except that the former lets us make use of the value of label to give a more informative error message.

This approach is often used when there a "balanced" symbols in the syntax. Consider the rule:

```
label     :  LABEL
          |  '<'  LABEL  '>'
                        { $$=$2; }
          |  '<'  error  '>'
                        { printf("Bad icon-label at line %d\n",
                          @1.first_line);
                        }
          ;
```

If the parser encounters something other than a LABEL after a '<', it will discard all tokens up to the next '>'. This technique can be useful for keeping brackets balanced during error-recovery.

In our case, keep in mind that any unidentifiable text after a valid LABEL is converted to an EXEC token by the lexer, and the EXEC token would swallow any subsequent '>'. So this rule in unlikely to trap any real errors, except something like < *keyword* >

In fact, it is likely to work against us, because we may not see another '>' at all, so we will miss out on a lot of input!

Q 3 Construct the following grammar

$E \rightarrow E + T / T$

$T \rightarrow T * F / F$

$F \rightarrow (E) / id$

a. Construct the first and follow sets

b. create LL(1) Parsing table

c. Construct LR Parsing table

d. Using LR parsing table , check whether id + id * id belongs to this grammar or not , Explain step by step procedure

Solu

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

$\longrightarrow$

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

(a)

FIRST(F) = { (,id}
FIRST(T′) = {*, ε}
FIRST(T) = { (,id}
FIRST(E′) = {+, ε}
FIRST(E) = { (,id}

FIRST(TE′) = { (,id}
FIRST(+TE′) = {+}
FIRST(ε) = {ε}
FIRST(FT′) = { (,id}
FIRST(*FT′) = {*}
FIRST(ε) = {ε}
FIRST((E)) = { (}
FIRST(id) = {id}

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

FOLLOW(T) = { +,), $ }

FOLLOW(T') = { +,), $ }

FOLLOW(F) = {+, *,), $ }

(b)

	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)		

(c)

LR(1) Parsing Table:

	id	+	*	(	)	$	E	T	F
0	shift 5			shift 4			1	2	3
1		shift 6				accept			
2		$E \to T$	shift 7		$E \to T$	$E \to T$			
3		$T \to F$	$T \to F$		$T \to F$	$T \to F$			
4	shift 5			shift 4			8	2	3
5		$F \to$ id	$F \to$ id		$F \to$ id	$F \to$ id			
6	shift 5			shift 4				9	3
7	shift 5			shift 4					10
8		shift 6			shift 11				
9		$E \to E + T$	shift 7		$E \to E + T$	$E \to E + T$			
10		$T \to T * F$	$T \to T * F$		$T \to T * F$	$T \to T * F$			
11		$F \to (E)$	$F \to (E)$		$F \to (E)$	$F \to (E)$			

(d)

Parsing: `id * (id + id)`.

step	Stack	Input	Action to Perform
1	0	id $*$ (id $+$ id)\$	shif 5
2	0 id 5	$*$ (id $+$ id)\$	$F \to$ id
3	0 F 3	$*$ (id $+$ id)\$	$T \to F$
4	0 T 2	$*$ (id $+$ id)\$	shif 7
5	0 T 2 $*$ 7	(id $+$ id)\$	shif 4
6	0 T 2 $*$ 7 (4	id $+$ id)\$	shif 5
7	0 T 2 $*$ 7 (4 id 5	$+$id)\$	$F \to$ id
8	0 T 2 $*$ 7 (4 F 3	$+$id)\$	$T \to F$
9	0 T 2 $*$ 7 (4 T 2	$+$id)\$	$E \to T$
10	0 T 2 $*$ 7 (4 E 8	$+$id)\$	shif 6
11	0 T 2 $*$ 7 (4 E 8 $+$ 6	id)\$	shif 5
12	0 T 2 $*$ 7 (4 E 8 $+$ 6 id 5	)\$	$F \to$ id
13	0 T 2 $*$ 7 (4 E 8 $+$ 6 F 3	)\$	$T \to F$
14	0 T 2 $*$ 7 (4 E 8 $+$ 6 T 9	)\$	$E \to E + T$
15	0 T 2 $*$ 7 (4 E 8	)\$	shif 11
16	0 T 2 $*$ 7 (4 E 8) 11	\$	$F \to (E)$
17	0 T 2 $*$ 7 F 10	\$	$T \to T * F$
18	0 T 2	\$	$E \to T$
19	0 E 1	\$	accept