INTRODUCTION TO FORMAL LANGUAGES AND AUTOMATA THEORY

Lecture Notes_8

Run-time Storage Organization

The run-time environment is the structure of the target computers registers and memory that serves to manage memory and maintain information needed to guide a programs execution process.

Run-time storage comes in blocks, where a byte is the smallest unit of memory. Four bytes form a machine word. Multibyte objects are bytes and given the address of first byte.

The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.

A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
This unused space due to alignment considerations is referred to as padding.

The size of some program objects may be known at run time and may be placed in an area called static.
The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Types of Runtime Environments –

1. **Fully Static:**
   Fully static runtime environment may be useful for the languages in which pointers or dynamic allocation is not possible in addition to no support for recursive function calls.
   - Every procedure will have only one activation record which is allocated before execution.

- Variables are accessed directly via fixed address.
- Little bookkeeping overhead; i.e., at most return address may have to be stored in activation record.
- The calling sequence involves the calculation of each argument address and storing into its appropriate parameter location and saving the return address and then a jump is made.

2. **Stack Based:**
   In this, activation records are allocated (push of the activation record) whenever a function call is made. The necessary memory is taken from the stack portion of the program. When program execution return from the function the memory used by the activation record is deallocated (pop of the activation record). Thus, the stack grows and shrinks with the chain of function calls.

3. **Fully Dynamic:**
   Functional language such as Lisp, ML, etc. use this style of call stack management. Silently, here activation record is deallocated only when all references to them have disappeared, and this requires the activation records to dynamically freed at arbitrary times during execution. Memory manager (garbage collector) is needed.
   The data structure that handles such management is heap an this is also called as Heap Management.

<u>Activation Record</u> –
The information which required during an execution of a procedure is kept in a block of storage called an activation record.
Information needed by a single execution of a procedure is managed using a contiguous block of storage called “activation record”.

An activation record is allocated when a procedure is entered and it is deallocated when that procedure is exited. It contains temporary data, local data, machine status, optional access link, optional control link, actual parameters and returned values.

- **Program Counter (PC) –** whose value is the address of the next instruction to be executed.
- **Stack Pointer (SP) –** whose value is the top of the (top of the stack, ToS).
- **Frame pointer (FP) –** which points to the current activation.

Uses

- It is used to store the current record and the record is been stored in the stack.
- It contains return value. After the execution the value is been return.
- It can be called as return value.

Parameter

- It specifies the number of parameters used in functions.

Local Data

- The data that is been used inside the function is called as local address

Temporary Data

- It is used to store the data in temporary variables.

Links

- It specifies the additional links that are required by the program.

Status

- It specifies the status of program that is the flag used.

PARSER

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens, interactive commands, or program instructions and breaks them up into parts that can be used by other components in programming.

A parser usually checks all data provided to ensure it is sufficient to build a data structure in the form of a parse tree or an abstract syntax tree.

In order for the code written in human-readable form to be understood by a machine, it must be converted into machine language. This task is usually performed by a translator (interpreter or compiler). The parser is commonly used as a component of the translator that organizes linear text in a structure that can be easily manipulated (parse tree). To do so, it follows a set of defined rules called “grammar”.

The overall process of parsing involves three stages:

1. **Lexical Analysis:** A lexical analyzer is used to produce tokens from a stream of input string characters, which are broken into small components to form meaningful expressions. A token is the smallest unit in a programming language that possesses some meaning (such as +, -, *, “function”, or “new” in JavaScript).
2. **Syntactic Analysis:** Checks whether the generated tokens form a meaningful expression. This makes use of a context-free grammar that defines algorithmic procedures for components. These work to form an expression and define the particular order in which tokens must be placed.
3. **Semantic Parsing:** The final parsing stage in which the meaning and implications of the validated expression are determined and necessary actions are taken.

A parser's main purpose is to determine if input data may be derived from the start symbol of the grammar. If yes, then in what ways can this input data be derived? This is achieved as follows:

- **Top-Down Parsing:** Involves searching a parse tree to find the left-most derivations of an input stream by using a top-down expansion. Parsing begins with the start symbol which is transformed into the input symbol until all symbols are translated and a parse tree for an input string is constructed. Examples include LL parsers and recursive-descent parsers. Top-down parsing is also called predictive parsing or recursive parsing.
- **Bottom-Up Parsing:** Involves rewriting the input back to the start symbol. It acts in reverse by tracing out the rightmost derivation of a string until the parse tree is constructed up to the start symbol This type of parsing is also known as shift-reduce parsing. One example is an LR parser.

Parsers are widely used in the following technologies:

- Java and other programming languages.
- HTML and XML.
- Interactive data language and object definition language.
- Database languages, such as SQL.
- Modeling languages, such as virtual reality modeling language.
- Scripting languages.
- Protocols, such as HTTP and Internet remote function calls.

LR Parsers

LR Parser is a class of Bottom-Up Parser that is used to parse Context-Free Grammars. LR Parsing is known as LR (K) parsing where

- L represents Left to Right Scanning of Input
- R represents Rightmost Derivation
- K is the number of input symbols of Look ahead that are used in developing parsing decisions.

LR parser is a shift-reduce parser that creates the use of deterministic finite automata, identifying the collection of all applicable prefixes by reading the stack from bottom to top.

It decides what handle, if any, is feasible. An achievable prefix of a right sentential form is that prefix that includes a handle, however no symbol to the right of the handle.

Thus, if a finite-state machine that recognizes viable prefixes of the right sentential forms is assembled, it can be used to influence the handle selection in the shiftreduce Parser.

Because the LR parser creates the use of a DFA that identifies viable prefixes to manage the selection of handles, it should maintain track of the states of the DFA. Therefore, the LR parser stack includes two kinds of symbols− state symbols can identify the states of the DFA and grammar symbols.

The parser begins with the initial state of a DFA, I_0 on the stack. The parser operates by considering the next input symbol 'a' and the state symbol I_i on the top of the stack.

If there is a transition from the state I_i on 'a' in the DFA going to state I_j therefore the symbol a is followed by state symbol I_j onto the stack.

If there is no transition from I_i on a in the DFA, and if the state I_i on the top of the stack identifies, when listed a viable prefix that includes the handle A → a, thus the parser gives out the reduction by popping α and pushing A onto the stack.

This is similar to developing a backward transition from I_i on α in the DFA and then developing a forward transition on A. Each shift action of the parser correlates to a transition on a terminal symbol in the DFA.

Thus, the current state of the DFA and the next input symbol determines whether the parser shifts the next input symbol or goes for reduction.

If it can produce a table mapping each state and input symbol pair as either "shift", "reduce", "accept", or "error", we obtain a table that can be used to manage the parsing phase. Such a table is known as the parsing "action" table.

Difference between LL and LR parser

LL Parser includes both the recursive descent parser and non-recursive descent parser. Its one type uses backtracking while another one uses parsing table. Theses are top down parser.

Example: Given grammar is

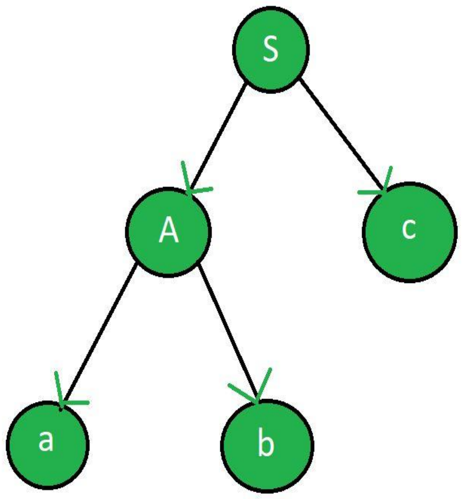
S -> Ac

A -> ab

where S is start symbol, A is non-terminal and a, b, c are terminals.

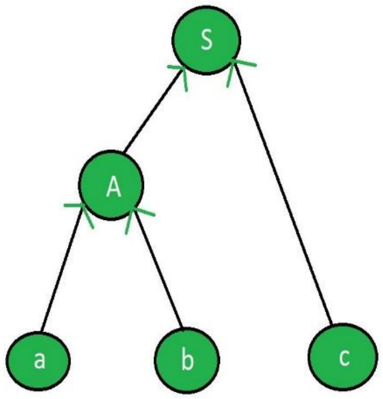
Input string: abc

Parse tree generated by LL parser:



LR Parser is one of the bottom up parser which uses parsing table (dynamic programming) to obtain the parse tree form given string using grammar productions.

Example: In the above example, parse tree generated by LR parser:



Difference between LL and LR parser:

LL Parser	LR Parser
First L of LL is for left to right and second L is for leftmost derivation.	L of LR is for left to right and R is for rightmost derivation.

LL Parser	LR Parser
It follows the left most derivation.	It follows reverse of right most derivation.
Using LL parser parser tree is constructed in top down manner.	Parser tree is constructed in bottom up manner.
In LL parser, non-terminals are expanded.	In LR parser, terminals are compressed.
Starts with the start symbol(S).	Ends with start symbol(S).
Ends when stack used becomes empty.	Starts with an empty stack.
Pre-order traversal of the parse tree.	Post-order traversal of the parser tree.
Terminal is read after popping out of stack.	Terminal is read before pushing into the stack.
It may use backtracking or dynamic programming.	It uses dynamic programming.

LL Parser	LR Parser
LL is easier to write.	LR is difficult to write.
Example: LL(0), LL(1)	**Example:** LR(0), SLR(1), LALR(1), CLR(1)