Brief Article

The Author

January 11, 2006

1 Top down parsing

The purpose of this section is to introduce the basic ideas behind top-down parsing and show how to construct an efficient non-backtracking form of top-down parser called a predictive parser. In the process, we construct a parse tree for an input string from the root and create the nodes of the parse tree in pre-order.

There are some catch phrases used through out this section, predictive parsing and recursive descent. Also, the term backtracking is used significantly. It is said that a top-down LL(1) parser seldom needs backtracking for a context free grammar, and tends to be inefficient on context sensitive grammars.

A left-recursive grammar can cause a a recursive descent parser, even one with backtracking to go into an infinite loop.

Predictive Parser Qualities:

- Eliminates left recursion
- left factors
- Proper alternatives are detected by examining the first symbols derives.

Transition Diagrams for Predictive Parsers

Differences between lexical analyzers and predictive parser transition.

- One diagram for each non-terminal
- Labels are both tokens and non-terminals
- Transition on non-terminal equates to a procedure call

To apply a transition diagram:

- Apply left recursion removal
- Left factor the grammar
- Create an initial and final return state
- For each production $A \rightarrow X_1, X_2, ..., X_n$ create a path from the initial to the final state with edges labeled $X_1, X_2, ..., X_n$.

What happens for a predictive parser which works off the transition diagram?

1. It begins in the start state for the start symbol.
2. After some actions, the parser arrives in state s.
   - Possibility: There is an edge to t labeled a if the next input symbol is a, then the transition to t occurs.
   - There is an edge to t labeled by non-terminal A.
     - The parser goes to the start state of A without moving the input cursor.
     - If a final state in A is reached, the transition is made.
   - There is an edge to t labeled by ϵ .
     - Transition is made without advancing the input cursor.

"A predictive parsing program based on a transition diagram attempts to match terminal symbols against the input, and makes a recursive procedure call whenever it has to follow an edge labeled by a nonterminal."

- For nondeterminism this does not work
- For determinism this works

Non-recursive predictive parsing A stack may be used explicitly to build a non-recursive predictive parser. Problems exist in determining the production to be applied. Solutions to these problems include parse table look up. Table-driven predictive parser (reference the diagram, page 184 "A table driven predictive parser.").

2 LL(1)

$LL(1)$ is a left to right scan of input Leftmost de

Function of boolean

All functions are boolean.

Example: $A \rightarrow aBa|C|\epsilon$ this a generic case rule. There is function that implements this rule. There is a selection set associated with A ($SS(A)$). Assumption ??? function A: boolean is this syntax rule satisfied. If next token equals 'a' (if nt = 'a') then do the following: get (nt) consume (a) if (B) then if nt = 'a' then consume (a) A := true else error A:= false. else A:= false else if C then A:= true else if $nt \in SS(A)$ then A := true else error A:=false

Note that all consumes call the lexical analyzer.

Note about if-else structures. In cases of if a if b if c is equivalent to an and, and if else if is analogous to an or. Reference the book on first, follow and recursive decent parsing.

Compiler-compilers came out of bottom-up compilers. Donald Knuth made this contribution to comp sci. Optimizations comes out of bottom-up compilers. Produce the selection sets and there is a full precise specification of the grammar.

Example grammar:

2:30:35

reference 2:32 for E1

next page 2:37:35

function T1: boolean reference 2:40

Tracing of the example function:

example finished at 3:11.

Knuth and argument.

There is a viable prefix property. Parses until the first error.

3 September 28

note at 2:10

Selection monitor at 2:13

$C \rightarrow E|E| <$

note 2:24

Palindrome problem 2:34

Slightly altered grammar 2:37

Solution 2:45

Delimiter to the palindrome problem is the semi-colon (;).

What the expression looks like: 2:50

Analyze language generated from E, 2:55

Trouble with comma's 3:02

Structured stand point 3:06

Error recovery and begin semantic analysis. Syntax analysis should be done by the end of the weekend.