

TOP DOWN PARSER BUILT IN JAVA

Akash Kulkarni
4JC11CS006

Anurag Kakati
4JC11CS015

May 8, 2014

1 Abstract

A new approach to designing system level softwares such as parsers using a high level programming language, namely Java, is presented in this paper. Parsing or syntactic analysis is the process of analysing a string of symbols, either in natural language or in computer languages, according to the rules of a formal grammar. A parser is a software component that takes input data and builds a parse tree, abstract syntax tree or other hierarchical structure giving a structural representation of the input, checking for correct syntax in the process. The parser is often preceded by a separate lexical analyser, which creates tokens from the sequence of input characters. Top-down parsing is a parsing strategy where one first looks at the highest level of the parse tree and works down the parse tree by using the rewriting rules of a formal grammar. An LL parser is a type of parser that does top-down parsing by applying each production rule to the incoming symbols, working from the left-most symbol yielded on a production rule and then proceeding to the next production rule for each non-terminal symbol encountered. The parser implemented is an LL(1) parser where the (1) signifies the parser reads ahead one token at a time.

2 Top Down Parsers

An example for a typical top down parser is a Recursive-Descent parser. A recursive-descent parsing program consists of a set of procedures for each non-terminal. Execution begins with the procedure for the start symbol, which halts and

announces success if its procedure body scans the entire input string. The limitations of a recursive-descent parser are:

- A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.
- A recursive-descent parser has exponential time complexity.

To eliminate the problem of exponential time complexity, LL(1) parsers were introduced.

2.1 First and Follow

The construction of top-down parser is aided by two functions, FIRST and FOLLOW, associated with a grammar G .

Define $FIRST(\alpha)$, where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α . If $\alpha \xRightarrow{*} \epsilon$, then ϵ is also in $FIRST(\alpha)$. To compute $FIRST(X)$ for all grammar symbols X , apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then $FIRST(X) = \{X\}$
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $FIRST(X)$ if for some i , a is in $FIRST(Y_i)$, and ϵ is in all of $FIRST(Y_1), \dots, FIRST(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \xRightarrow{*} \epsilon$. If ϵ is in $FIRST(Y_j)$ for all $j = 1, 2, \dots, k$, then add ϵ to $FIRST(X)$.
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$

Define $FOLLOW(A)$, for a non-terminal A , to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \xRightarrow{*} \alpha A a \beta$, for some α and β .

To compute $FOLLOW(A)$ for all nonterminals 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 end marker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in $FIRST(\beta)$ except ϵ is in $FOLLOW(B)$

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where $\text{FIRST}(\beta)$ contains ϵ , then everything in $\text{FOLLOW}(A)$ is in $\text{FOLLOW}(B)$.

2.2 Parsing via construction of parsing table

Algorithm : Construction of parsing table.

INPUT : Grammar G

OUTPUT : Parsing table M

METHOD : For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in $\text{FIRST}A$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

Algorithm for non-recursive predictive parsing is given below:

Algorithm : Table-driven predictive parsing.

INPUT : A string w and a parsing table M for grammar G .

OUTPUT : If w is in $L(G)$, a leftmost derivation of w ; otherwise, an error indication.

METHOD : Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above $\$$.

```

set  $ip$  to point to the first symbol of  $w$ ;
set  $X$  to the top stack symbol;
while ( $X \neq \$$ ) {
    if ( $X$  is  $a$ ) pop the stack and advance  $ip$ ;
    else if ( $X$  is a terminal)  $error()$ ;
    else if ( $M[X, a]$  is an error entry)  $error()$ ;
    else if ( $M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$ ) {
        output production  $X \rightarrow Y_1 Y_2 \cdots Y_k$ ;
    }
}

```

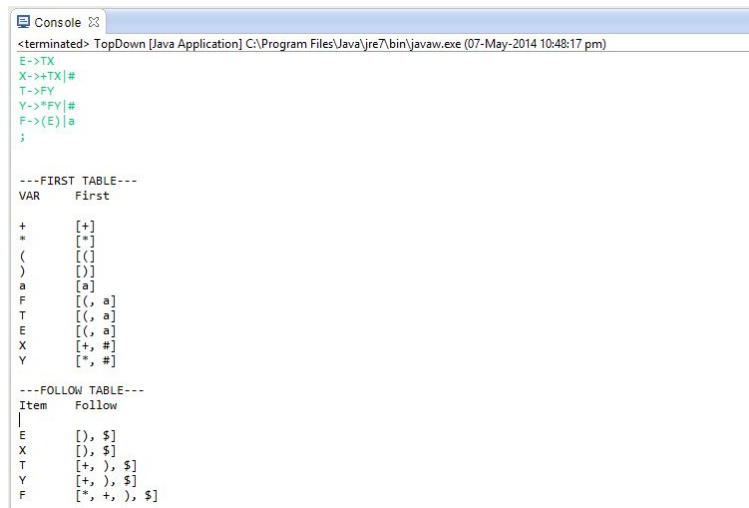
```

        pop the stack;
        push  $Y_k, Y_{k-1}, \dots, Y_1$  onto the stack, with  $Y_1$  on top;
    }
    set  $X$  to top stack symbol;
}

```

The parser implemented follows the above mentioned algorithms. First, Follow, and the Parsing table implementations mainly use the dictionary data-type available in Java. The parsing action implementation uses the traditional stack data structure for storing the non-terminals that are encountered during parsing. The dictionary data-type is an extended version of hashing and hence the efficiency of the retrieval operation from those data-types is in $O(n)$.

3 Simulation Results



```

Console
<terminated> TopDown [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (07-May-2014 10:48:17 pm)

E->TX
X->+TX|#
T->*FY
Y->*FY|#
F->(E)|#
;

---FIRST TABLE---
VAR      First
+         [+]
*         [*]
(         [(]
)         [)]
a         [a]
F         [(, a]
T         [(, a]
E         [(, a]
X         [+ , #]
Y         [+ , #]

---FOLLOW TABLE---
Item      Follow
|
E         [), $]
X         [), $]
T         [+ , ), $]
Y         [+ , ), $]
F         [+ , + , ), $]

```

4 Conclusion

The advantage of using a high level language like Java is clearly enumerated by the design and execution of the program. It is easier to design the parser in Java than most other programming languages and advanced concepts like

```

Console
<terminated> TopDown [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (07-May-2014 10:48:17 pm)

---PARSE TABLE---
+      *      (      )      a      $

E |      -      -      E->[TX]      -      E->[TX]      -
X |      X->[+TX]      -      -      X->[#]      -      X->[#]
T |      -      -      T->[FY]      -      T->[FY]      -
Y |      Y->[#]      Y->[*FY]      -      Y->[#]      -      Y->[#]
F |      -      -      F->[(E)]      -      F->[a]      -

Enter String : a+a*a

Moves made by a predictive parser on input :a+a*a

-----
MATCHED      |      STACK      INPUT      ACTION
-----
      [E, $]      a+a*a$
      [T, X, $]      a+a*a$      output E->[TX]
      [F, Y, X, $]      a+a*a$      output T->[FY]
      [a, Y, X, $]      a+a*a$      output F->[a]
      [Y, X, $]      +a$a$      match a
      [X, $]      +a$a$      output Y->[#]
      [+ , T, X, $]      +a$a$      output X->[+TX]
      [T, X, $]      a$a$      match +
      [F, Y, X, $]      a$a$      output T->[FY]
      [a, Y, X, $]      a$a$      output F->[a]
      [Y, X, $]      "a$      match a
      [*, F, Y, X, $]      "a$      output Y->[*FY]
      [F, Y, X, $]      a$      match *
      [a, Y, X, $]      a$      output F->[a]
      [Y, X, $]      $      match a
      [X, $]      $      output Y->[#]
      [$]      $      output X->[#]

---SUCCESSFULL---

```

memoization and advanced datatypes like ArrayLists make for a more efficient parsing of input data.

5 References

1. Wikipedia
2. Compilers Principles, Techniques, and Tools - Alfred V Aho, Monica S Lam, Ravi Sethi, Jeffery D Ullman
3. HeadFirst Java