Compiler Design ReferencePoint Suite

Table of Contents

Point 2: Parsing Techniques

Surbhi Singhal

Parsing is a compiler phase in which a syntax tree for an input string is produced as output. A parser program parses a string that consists of tokens generated by the lexical phase of a compiler and generates a syntax tree as the output. This tree is also known as a parse tree.

There are two types of strategies for parsing input strings a bottom-up parsing technique and a top-down parsing technique. In the bottom-up parsing technique, a parse tree for an input string is built from leaves to root; in top-down parsing, a parse tree is built from root to leaves.

This ReferencePoint describes various parsing techniques, such as shift reduce, operator precedence, top-down, and predictive parsers, and how these techniques relate to one another.

Shift-Reduce Parsing

A shift-reduce parser uses the bottom-up parsing technique. For example, you have a grammar with productions numbered as:

1. X -> gGiHk

2. G -> Gh | h

3. H -> j

You also have string ghhijk. In the bottom-up construction of a parse tree, the string is reduced to the start symbol of a grammar.

To parse the input string using the shift-reduce parsing method:

1. Scan the string from the left for sub strings that match the right side of any production rule. The leftmost h in input string ghhijk matches the right side of production rule 2. Replace the leftmost h of the string with symbol G from production rule 2 to obtain string gGhijk.

2. Replace the leftmost GH of the string with production rule 2. The string obtained is gGijk.

3. Replace the leftmost j of the string with H from production rule 3. String gGiHk is obtained. The entire string gGiHk matches the right side of production rule 1.

4. Replace the string with X. The input string ghhijk is reduced to the start symbol of the grammar.

In shift-reduce parsing, a substring that matches the right side of a production is replaced by the left side of the production. These replacements are called reductions. The string ghhijk is reduced to the start symbol of the grammar in a sequence of four reductions.

Concept of Handles

A handle in a right sentential form is defined as a production of the form G -> combined with a position of substring in . In this combination, you can replace by G, to obtain a string that can be reduced to the start symbol of a grammar. For example, if X is the start symbol of a grammar such that:

X -> G ->

The production G -> , with the position where is substituted, is a handle of string . The form is right sentential. As a result, contains only terminal symbols. For example, a grammar contains productions numbered as:

1. X -> gGiHk

2. G -> Gh | h

3. H -> j

The ghhijk string, a right sentential form of the grammar, contains a handle G -> h at position 2.

Every right-sentential form of an unambiguous grammar contains exactly one handle. Reducing a right-sentential string to the start symbol of the grammar is similar to finding the rightmost derivation in reverse. This type of reduction is called the canonical reduction of the input string. Handles enable you to perform this reduction. To find a canonical reduction sequence for a right-sentential form , locate a handle hn in and replace it with the left-side of production G -> hn.

For example, you have a grammar with productions numbered as:

1. X -> X + X

2. X -> X * X

3. X -> (X)

4. X -> iden

The input string is iden1 + iden2 * iden3. Figure 2-2-1 shows how to parse the string iden1 + iden2 * iden3 to the start symbol X of the grammar by locating handles:

Input String	Handle	Reduced String
iden1 + iden2* iden3	iden1	X-> iden
X + iden2* iden3	iden2	X-> iden
X + X* iden3	iden3	X->iden
X + X* X	X* X	X-> X* X
X + X	X + X	X-> X + X
X		

Figure 2-2-1: Parsing the String iden1 + iden2 * iden3

Implementing a Shift-Reduce Parser

To implement shift-reduce parsing, you need to locate handles in an input string and select a production with the same right side. You can implement a shift-reduce parser with a push-down store and an input buffer. The $ symbol is used to mark the bottom of a push-down store and the right end of the input. Figure 2-2-2 shows the initial configuration of the push-down store and the input buffer:

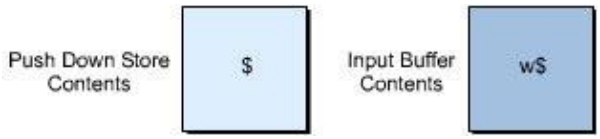

Figure 2-2-2: Configuration of the Push-Down Store and the Input Buffer Before Parsing
In Figure 2-2-2, w denotes the input string. The parser works by shifting symbols from the string to a push-down store until a handle is detected on the top of the push-down store. The right side of a detected handle is reduced to the left side of the matched production. Repeat this process until an error is reported or the push-down store contains the start symbol of the grammar, and the input buffer is empty. Figure 2-2-3 shows the configuration of a push-down store after parsing the input string:

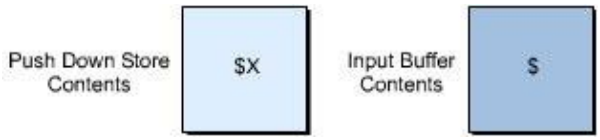

Figure 2-2-3: Configuration of the Push-Down Store and the Input Buffer After Parsing
Actions of a shift-reduce parser in parsing an input string iden1 + iden2 * iden3 derived from grammar with productions as:

1. X -> X + X
2. X -> X * X
3. X -> (X)
4. X -> iden

The grammar is shown in Figure 2-2-4:

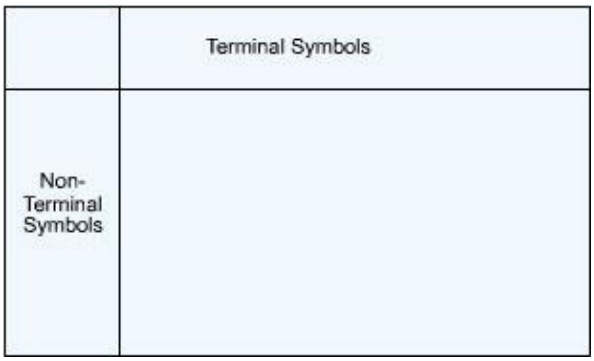

Figure 2-2-4: Actions of the Shift-Reduce Parser
The basic operations performed by a shift-reduce parser are:

- Shift: Specifies that the next input symbol in the input buffer is to be shifted to a push-down store.

- Reduce: Recognizes and replaces the left end of the handle with the right side of an appropriate production. The parser performs a Reduce move when a handle is detected at the top of a push-down store.

- Accept: Specifies the successful completion of parsing.

- Error: Specifies a syntax error. The parser invokes an error recovery subroutine.

Building a Parse Tree

You can build a parse tree for the input string while performing the shift-reduce parsing. A pointer that points to a tree is associated with each symbol in a push-down store. The root of the pointer is the symbol itself. When you complete parsing, the start symbol on the top of a push-down store will have the entire parse tree associated with it. To construct the parse tree:

- Construct a single-node tree labeled x, if you shift x in a push-down store.

- Reduce a set of symbols to a single symbol. For example, to reduce T1, T2, T3,.....Tn to G, a new node G is created and T1 T2 T3..... Tn, become children of node G.

Figure 2-2-5 shows how to construct a parse tree for string iden1 + iden2 * iden3:

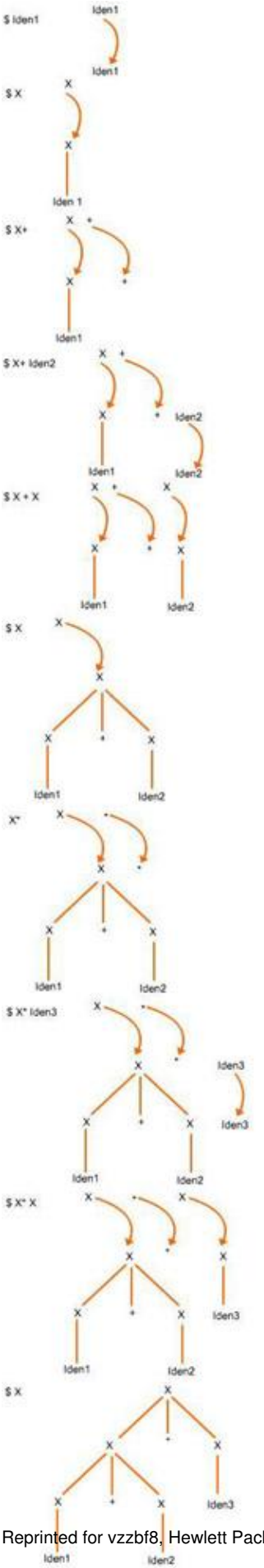
$ Iden1
Iden1
Iden1
$ X
X
X
Iden 1
$ X+
X
+
X
+
Iden1
$ X+ Iden2
X
+
X
+
Iden2
Iden1
Iden2
$ X + X
X
+
X
X
+
X
Iden1
Iden2
$ X
X
X
X
+
X
Iden1
Iden2
X*
X
*
X
*
X
+
X
Iden1
Iden2
$ X* Iden3
X
*
X
*
Iden3
X
+
X
Iden3
Iden1
Iden2
$ X* X
X
*
X
X
*
X
X
+
X
Iden3
Iden1
Iden2
$ X
X
X
+
X
X
+
X
Iden3
Iden1
Iden2

Figure 2-2-5: Construction of a Parse Tree

Operator-Precedence Parsing

Operator-precedence parsing is a bottom-up parsing method used to parse an operator grammar. An operator grammar is one in which the right side of production does not contain two adjacent non-terminal symbols and no non-terminal derives a symbol. For example, a grammar with productions numbered as follows, is not an operator grammar because it contains two adjacent non-terminals in the right side of production rule 1:

1. G -> G B G | iden

2. B -> * | - | / | +

The grammar is converted to an operator grammar by replacing non-terminal B with its alternates in production, G -> G B G. The resulting grammar is an operator grammar with the following production:

```
G -> G * G | G - G | G / G | G + G | iden
```

You need to define precedence relations among various symbols of a grammar to locate handles in an operator-precedence parsing method. These relations define the precedence between operators. The precedence relations used in an operator-precedence parsing method are:

- g <. h: Indicates that g holds lower precedence than h

- g .> h: Indicates that g holds higher precedence than h

- g = h: Indicates that g holds equal precedence to h

Here, g and h denote symbols of a grammar.

Operator-Precedence Relations Based on Precedence and Associativity of Operators

You can define operator-precedence relations between the terminal symbols of a grammar on the basis of the precedence and associativity relations between various operators. Rules for defining precedence relations are:

1. If a grammar contains operators o1 and o2 and operator o1 has higher precedence than operator o2, the precedence relation o1.> o2 and o2 <.o1 holds between o1 and o2.

2. If operators o1 and o2 have the same precedence, then the precedence relation is decided based on the associativity of the operators. If the operators are right-associative, the precedence relation o1 <. o2 and o2 .> o1 is valid for the operators. The precedence relation o1.> o2 and o2 <. o1 is valid if the operators are left-associative.

3. If $ marks the left and right ends of an input string and o is an operator symbol, then $ <. o and o .> $.

Table 2-2-1 lists operator-precedence relations for a grammar GO with production rule:

```
G -> G * G | G - G | G / G | G + G | iden
```

Table 2-2-1: Operator-Precedence Relations Between Terminal Symbols Based on Precedence and Associativity of Operators

	*	/	+	-	iden	$
*	.>	.>	.>	.>	<.	.>
/	.>	.>	.>	.>	<.	.>
+	<.	<.	.>	.>	<.	.>
-	<.	<.	.>	.>	<.	.>
iden	.>	.>	.>	.>		.>
$	<.	<.	<.	<.	<.	

In the above table, * and / hold higher precedence than + and -. Symbols * and / are of equal precedence and are left associative; symbols + and - are also of equal precedence and left associative. The rows and columns of the table list all terminal symbols of the grammar. The $ symbol, which is a delimiter symbol, is of lower precedence to operators of a grammar. This ensures the detection of all handles in an input string.

Defining the Operator-Precedence Grammar

You can define precedence relations among terminal symbols of a grammar based on an operator-precedence grammar. A grammar that does not contain any ambiguity and whose productions do not contain a symbol on the right side is called an operator-precedence grammar. Rules to define precedence relations among various terminal symbols in an operator-precedence grammar are:

- If a grammar contains a production G -> g H w, and H derives a string alpha b where alpha is either a single non-terminal or a symbol, then g holds lower precedence than b. That is, g <. b.
- If a grammar contains a production G -> g H b delta, where H is either a single non-terminal or a symbol, then g holds equal precedence to b. That is, g = h.
- If a grammar contains a production G -> H b w, and H derives a string alpha g delta where delta is either a single non-terminal or a symbol, then g holds greater precedence than b. That is, g .> b.

For example, a grammar with the following production is free, ambiguous grammar:

```
G -> G - G | G / G | (G)  | iden
```

Modified productions after removing ambiguity from the grammar are:

1. G -> G - H | H
2. H -> H / K | K
3. K -> [G] | iden

This grammar is unambiguous operator grammar and does not contain symbol. As a result, it is a valid operator-precedence grammar.

The right side of production 1 contains a non-terminal G to the left of terminal -. As a result, according to rule 1, - holds lower precedence than all symbols that are the first terminal symbols in any string derivable from G. Similarly, you can define precedence relations for all terminal symbols in the grammar. Table 2-2-2 lists precedence relations for the terminal symbols of the grammar:

Table 2-2-2: Operaor-Precedence Relations Between Terminal Symbols Based on Operator-Precedence Grammar

	-	/	[	]	iden	$
-	.>	<.	<.	.>	<.	.>
/	.>	.>	<.	.>	<.	.>
[	<.	<.	<.	=	<.	.>
]	.>	.>		.>	<.	.>
Iden	.>	.>	.>	.>		.>
$	<.	<.	<.	<.	<.	

To define the precedence relations in an operator-precedence grammar, you need to find the first and last terminal symbols derivable from non-terminal symbols in that grammar. LEADING (G) is a function that computes the first terminal symbol derivable from G, based on the following rules:

- If a grammar contains a production G -> alpha g and alpha is either a single non-terminal symbol or a symbol, then LEADING (G) includes g.
- If a grammar contains a production G -> H delta and LEADING (H) contains g, then g is included in LEADING (G).

TRAILING (G) is a function that computes the last terminal symbol derivable from G based on the following rules:

- If a grammar contains a production G -> alpha g and is either a single nonterminal or a symbol, then TRAILING (G) includes g.
- If a grammar contains a production G -> delta H and TRAILING (H) contains g, then g is also included in TRAILING (G).

Locating Handles in an Operator-Precedence Parsing Method

Precedence-relations in an operator-precedence grammar enable you to locate the handle in an input string. In the operator-precedence parsing method, a string delimited between <. and .> forms a handle. Symbol <. acts as the left delimiter symbol of a handle; symbol .> is the right-end delimiter of a handle. The symbol = may or may not appear between the left-end and right-end delimiter symbols. For example, Table 2-2-3 lists the steps for parsing the input string iden - iden / iden, based on the precedence relations of grammar GO:

Table 2-2-3: Parsing Input String

Push-Down Store Contents	Precedence Relation	Input String
$	<.	iden - iden / iden $
$ <. Iden	.>	- iden / iden $
$ <. G	-	- iden / iden $
$ <. G -	<.	Iden / iden $
$ <. G - <. iden	.>	/ iden $
$ <. G - <. G	-	/ iden $
$ <. G - <. G /	<.	iden $
$ <. G - <. G / <. Iden	.>	-
$ <. G - <. G / G .>	-	-
$ <. G - G .>	-	-
$ G	-	-

The string is parsed when the push-down store contains only the start symbol of the grammar. Whenever the <. symbol is encountered during parsing, the next input symbol is shifted in the push-down store. When the .> symbol is encountered, a handle is detected. The push-down store is searched for the <. symbol. The string between <. and .> symbols is a handle and is reduced according to an appropriate production in the grammar.

Top-Down Parsing

In the top-down parsing method, the parser parses the input string by constructing a parse tree starting from the root and proceeding to the leaves. This means that the parser finds the leftmost derivation of the input string. For example, a grammar contains productions numbered as follows:

1. X -> ikG

2. G -> gh | g

To parse the input string igk using the top-down parsing method:

1. Create a node labeled X, where X is the start symbol of the grammar. Figure 2-2-6 shows the parse tree with a single node, labeled X:

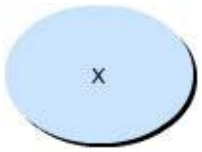

Figure 2-2-6: Parse Tree with a Single Node

2. Expand X according to production 1. The symbols in the right side of the production become the children of root node X. The input pointer points to the input string. Initially, the pointer points to symbol i, which is the leftmost symbol of the string. Figure 2-2-7 shows the parse tree:

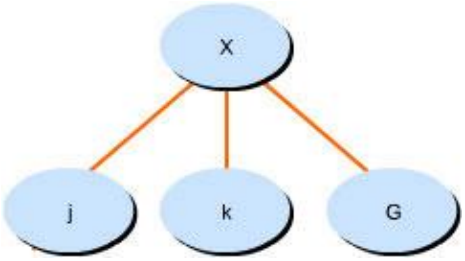

Figure 2-2-7: Parse Tree When X is Expanded

3. The leftmost leaf, a terminal symbol i, matches the first symbol of the input string, so advance the input pointer to point to the second symbol of the input string. The next leaf, k, matches the second symbol of the string. The input pointer is advanced to point to the third symbol of the input string.

4. Expand the third leaf, which is a non-terminal, using the production rule 2. Expand G using the first alternate for G to obtain the parse tree, as shown in Figure 2-2-8:

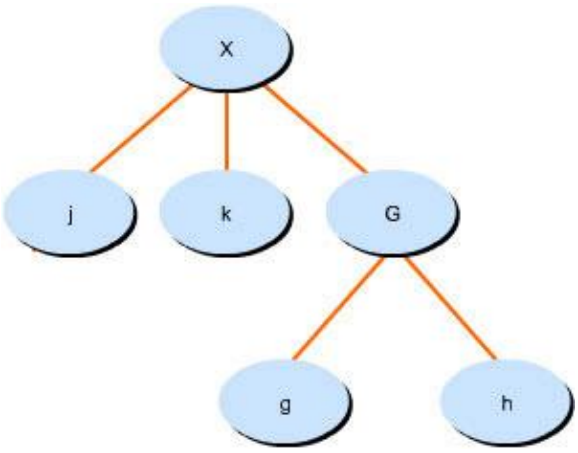

Figure 2-2-8: Expanding Leaf Node G

5. Match the third symbol in the input string with the leaf labeled g. There are no more symbols left in the input string, but there is one more leaf. As a result, a failure is reported and step 4 is undone. The other alternate for G—that is, G -> g—is considered, and the input pointer is repositioned at the third symbol of the input string. Figure 2-2-9 shows the parse tree obtained:

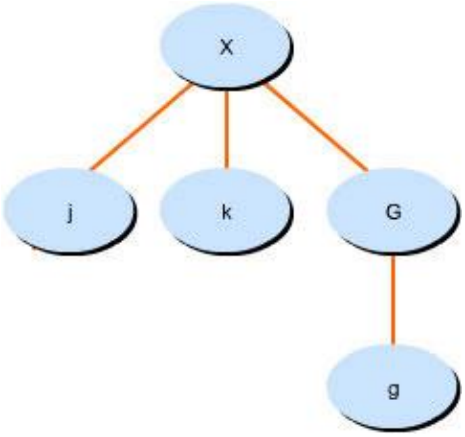

Figure 2-2-9: Expanding G Using Alternate Production

6. Match the leaf labeled g with the third input symbol of the input string. The symbols match and the parsing of input string ikG is completed.

Problems with Top-Down Parsing

A grammar that contains a production of the form G -> Gw is called a left-recursive grammar, where w is a sequence of terminals and non-terminals. A top-down parser can result in an endless loop with a left-recursive grammar. If you try to expand a leaf labeled G, where G is a non-terminal symbol using production G -> Gw, the expansion might result in an infinite loop. A grammar used with a top-down parser should not be left-recursive.

Backtracking is the process of moving up the parse tree from a specific point to undo an action such as a symbol table entry. This involves considerable overhead. As a result, you should avoid backtracking in top-down parsers.

Selecting an inappropriate alternate to construct a parse tree leads to the rejection of the input string. For example, Figure 2-2-10 shows the parse tree constructed when you parse an input string ikgih with a grammar:

1. X -> ikGh

2. G -> g | gi

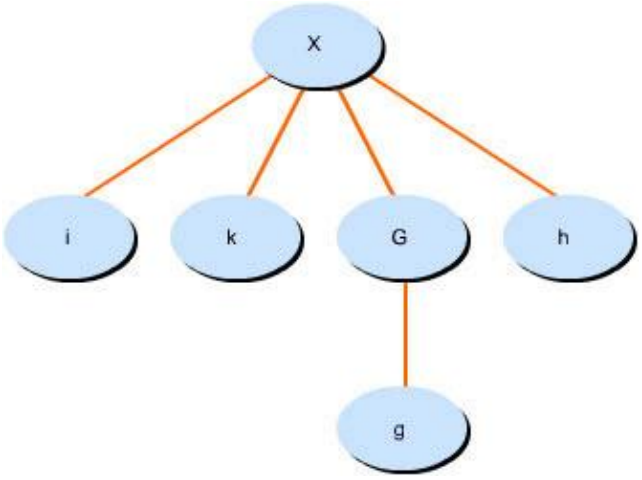

Figure 2-2-10: Parse Tree
The first three leaf nodes of the parse tree match the leftmost input string symbols. The fourth input string symbol, i, does not match leaf node h. This signifies that the production for X is incorrect and the parser will reject input string ikgih. On the other hand, if non-terminal G is expanded using production G -> gi, the parser accepts the input string.

Removing Left-Recursion in a Grammar

You can remove left-recursion from a grammar by modifying productions. For example, if there is a left-recursive grammar with production G -> G | , you can modify this grammar as:

```
G ->  G'
G' ->  G' |
```

Here, and are strings that consists of terminals and nonterminals; G' is a nonterminal symbol.

Modifying production rules does not change the sequence of strings that are derivable from G. Figure 2-2-11 shows the parse tree generated with the original production G -> G | :

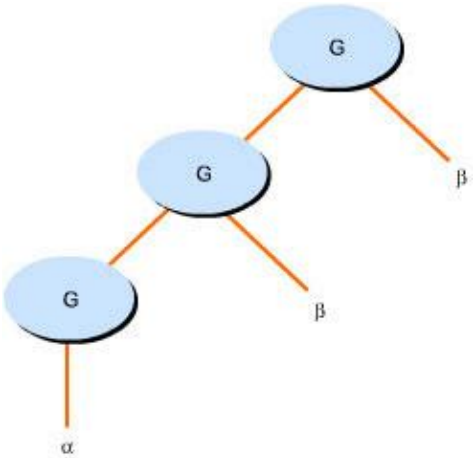

Figure 2-2-11: Parse Tree for a Left-Recursive Grammar
Figure 2-2-12 shows the parse tree with modified productions:

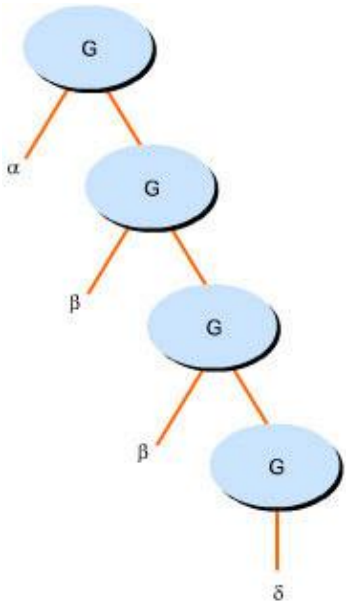

Figure 2-2-12: Parse Tree When Left-Recursion is Removed
For example, a left-recursive grammar contains productions numbered as follows:

1. K -> K * H | H
2. H -> H + A | A
3. A -> { K } | id

The grammar after modifying left-recursive productions is:

1. K -> HK'
2. K' -> * HK' |
3. H -> A H'
4. H' -> + A H' |
5. A -> { K } | iden

Top-Down Parsers Without Backtracking

You can eliminate backtracking by selecting an appropriate production from a grammar while expanding a nonterminal. The top-down parsing method that involves no backtracking is called a recursive-descent parsing method.

For example, in the derivation of the input string in a grammar with the following productions, the parser may use an incorrect alternate for X:

```
X -> aGb
X -> aKbp
```

If the parser uses an incorrect alternate, you have to backtrack the parser later. Left-factoring is the technique used to manipulate grammars that contain common leftmost prefixes for alternates. The production X -> w | contains the common prefix in its alternates. By applying left-factoring, productions become:

```
X ->   X'
X' -> w |
```

These productions do not contain common prefixes.

For example, the grammar for an if-then-else statement is:

```
S -> if C then S;
S -> if C then S else P;
```

The parser does not know which production it should use for S, so left-factoring is applied to productions. After left-factoring, the grammar becomes:

```
S -> If C then S K;
K -> ;
K -> else P;
```

When the parser encounters S, it expands S to If C then S K;. Then the parser uses an appropriate alternate for K to derive the input string.

Predictive Parsers

A predictive parser uses a push-down store, an input buffer, and a parsing table. The input buffer contains the string to be parsed and uses a $ symbol as the delimiter. Initially, a push-down store contains the delimiter symbol $ and the start symbol of the grammar. The entries in the parsing table define rules for parsing the input string. Figure 2-2-13 shows a predictive parser:

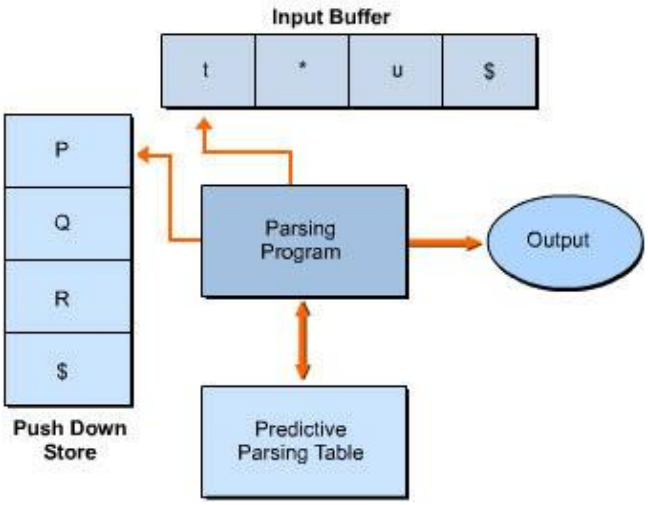

Figure 2-2-13: Structure of a Predictive Parser

The parsing program points to the topmost symbol in a push-down store and points to the leftmost symbol in the input buffer. These symbols determine the action of the parser so that it functions in accordance with the parsing table. Rules according to which a parser parses an input string are:

- Pop a symbol from a push-down store, if the topmost symbol in a push-down store and the leftmost symbol in the input string are equal. The next symbol in a push-down store becomes the topmost symbol.

- The parsing of the string is completed when the topmost symbol in the push-down store and the leftmost input string symbol are equivalent to the $ symbol.

- Look up the entry T [P, t] in the parsing table if the topmost symbol P in a push-down store is a nonterminal symbol. A push-down store symbol is replaced with the right side of the production if the production is mentioned in the table entry. If no entry is mentioned in the parsing table, an error is reported.

Computing FIRST and FOLLOW

Figure 2-2-14 shows the structure of a parsing table that defines the rules for parsing a specific input string. Terminal symbols of the grammar form the column headings and the non-terminal symbols form the row headings of the parsing table.

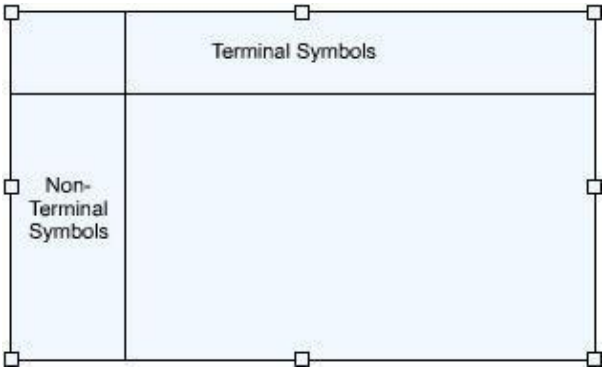

Figure 2-2-14: Structure of a Predictive Parsing Table

You can fill the entries of the parsing table according to the concept of FIRST and FOLLOW. Methods for calculating FIRST of a symbol G, FIRST (G) are:

- FIRST (G) is {G}, if G is a terminal symbol of a grammar.

- If G is a nonterminal symbol of a grammar, and the grammar contains a production of the form G -> bw, b is included in FIRST (G). The symbol is added to FIRST (G) if the grammar contains a production of the form G -> .

- If a grammar contains a production of the form G -> K_1 K_2 K_3..K_n, where K_1 K_2 K_3..K_n are nonterminal symbols, FIRST (G) is calculated by adding FIRST (K_1) to FIRST (G). If FIRST (K_1) derives , then add FIRST (K_2) also to FIRST (G). This is repeated for all nonterminal symbols K_1 K_2 K_3..K_n.

Methods for calculating FOLLOW (G), where G is a nonterminal symbol of a grammar, are:

- If a grammar contains a production G -> K , and is any grammar symbol other than , FOLLOW (K) is same as FIRST ( ) except for the symbol.

- If a grammar contains a production G -> K , where can be a symbol and derives , FOLLOW (K) includes symbols in FOLLOW (G).

- If K is the start symbol of a grammar, then FOLLOW (K) includes the $ symbol.

For example, a grammar GR contains productions numbered as follows:

1. G -> H G'

2. G' -> - H G' |

3. H -> K H'

4. H' -> / K H' |

5. K -> [G] | iden

To compute FIRST (A), where A is a symbol of the grammar:

```
FIRST (H) = FIRST (K) according to production number 3.
FIRST (K) = { [, iden }
```

As a result:

```
FIRST(G) = { [, iden }
FIRST (H) = { [, iden }
FIRST (H') = { /,  }
FIRST (G') = { -,   }
```

To compute FOLLOW (A), where A refers to the non-terminal symbols of grammar:

- FOLLOW (G): Computes FOLLOW (G). G is the start symbol of the grammar. As a result, symbol $ is in FOLLOW (G). From production K -> [G] | iden and rule 1 FOLLOW (G) contains {] }. As a result, FOLLOW (G) = { $,]).

- FOLLOW (G'): Computes FOLLOW (G'). FOLLOW (G') = FOLLOW (G) by applying rule 2 to production G -> H G'. As a result, FOLLOW (G') = { $,]).

- FOLLOW (H): Computes FOLLOW (H). FOLLOW (H) = FOLLOW (G) by applying rule 2 to production G -> H G'. Since G' is not equal to , so by rule 1, FOLLOW (H) also contains FIRST (G'). As a result, FOLLOW (H) = { -,], $ }.

- FOLLOW (H'): Computes FOLLOW (H'). FOLLOW (H') = FOLLOW (H) by applying rule 2 to production, H -> K H'.

- FOLLOW (K): Computes FOLLOW (K). FOLLOW (K) = FOLLOW (H) according to production H -> K H'. Since H' is not equal to , FOLLOW (K) also includes FIRST (H') according to rule 1. As a result, FOLLOW (K) = { -,], /, $ }.

Constructing Parsing Table for a Predictive Parser

When parsing an input string, a predictive parser takes an action depending on the entries of the parsing table. You can fill the entries in the parsing table depending on the concept of FIRST and FOLLOW. If a grammar includes production G -> w and FIRST (w) contains h, then the entry in the parsing table corresponding to G and h will be filled by production G -> w. Whenever the parser encounters G as the topmost stack symbol and h as the current symbol in the input buffer, it uses production G -> w to expand G. If w is equal to or w derives , and the current input symbol h, in the buffer, is in FOLLOW (G), enter the production G -> w in the entry corresponding to G and h in the parsing table.

Table 2-2-4 is a predictive parsing table that lists entries based on FIRST and FOLLOW values computed. FIRST (H) includes { [, iden }. So whenever you have G as the topmost symbol in a push-down store and [or iden, as the current input symbol, production G -> H G' is used to expand the topmost push-down store symbol.

Table 2-2-4: Parsing Actions of a Predictive Parser

-	-	/	[	]	iden	$
G			G -> H G'		G -> H G'	
G'	G' -> - H G'			G' ->		G' ->
H			H -> K H'		H -> K H'	
H'	H' ->	H' -> / K H'		H' ->		H' ->
K			K -> [G]		K -> iden	

Parsing an Input String

The actions taken by a predictive parser to parse input string iden - iden / iden, based on grammar GR and the parsing table, are listed in Table 2-2-5:

Table 2-2-5: Parsing an Input String by the Predictive Parsing Method

Push-Down Store Contents	Input Buffer	Action
$ G	iden - iden / iden $	-
$ G' H	iden - iden / iden $	G -> H G'
$ G' H ' K	iden - iden / iden $	H -> K H'
$ G' H' iden	iden - iden / iden $	K -> iden
$ G' K'	- iden / iden $	-
$ G'	- iden / iden $	K' ->
$ G' H -	- iden / iden $	G' -> - H G'
$ G' H	iden / iden $	-
$ G' H' K	iden / iden $	H -> K H'
$ G' H' iden	iden / iden $	K -> iden
$ G' H'	/ iden $	-
$ G' H' K /	/ iden $	H' -> * K H'
$ G' H' K	iden $	
$ G' H' iden	iden $	K -> iden
$ G' H'	$	-
$ G'	$	H' ->

$	$	G' ->

The parsing of the input string is completed when the push-down store and the input buffer contain only a delimiter symbol, $. The process of parsing in a predictive parser is the reverse of the process in a left-most derivation.

Related Topic

For related information on this topic, you can refer to Principles of Compiler Design