Summary of Top-Down Parsing Techniques

1. Main Concepts

The most important concept with top-down parsing is production-rule selection based on prediction. The main piece of information used to make such prediction is the computation of FIRST sets for the right-hand side of each production rule and the FOLLOW sets for nullable nonterminals.

The implementation of such top-down predictive parsing can be done in one of two ways. The method that directly uses the LL(k) parsing theory employs an LL(k) parsing table, where k usually equals 1, and a parsing stack. This method, however, is not easy to use for propagation semantic values. To make semantic value propagation easier during top down parsing, one can use the recursive-descent method, in which the stack mechanism underlying the execution of function calls is utilized, without the use of an explicit parsing stack.

2. Examples of LL(1) Parsing

In class, we worked on two examples. We go over these again with slightly revised production rules.

Example 1:

1. <program> → L
2. L → S L
3. L → S
4. S → var id
5. S → var id = <expr>
6. S → id = <expr>
7. S → document.write(<expr>)
8. <expr> → id
9. <expr> → num

The FIRST sets of rules 2 and 3 overlap, so do the FIRST sets of rules 4 and 5. This makes it impossible to predict which rule to apply based on the current token. We perform left factoring in an attempt to correct this problem. Rules 2 and 3 are changed to the following:

2’. L → S L’
3’. L’ → ε
3’’. L’ → L

Rules 4 and 5 are changed to the following:

4’. S → var id S’
5’. S’ → ε
5’’. S’ → = <expr>

We first determine nullable nonterminals, which are nullable={S’, L’}.

We then compute the FIRST sets for each non-ε rule, where 1 means rule 1’s right-hand side, and so on, just to make the notation short. Note that an ε-rule’s right hand side has an empty FIRST set and therefore cannot be decided whether to choose by using the FIRST set. It will use the FOLLOW set instead. A nullable nonterminal may have a nullable non-ε rule. In that case, both the FIRST set of the right-hand side, if nonempty, and the FOLLOW set of the nonterminal may need to be used to determine whether the rule is applied. We don’t have such a case in this example, however.

FIRST(8) = {id}, FIRST(9)={num}, FIRST(7) = {document.write}, FIRST(6) = {id},

FIRST(4’) = {var}, FIRST(5’’) = {=} , FIRST(2’) = FIRST(S) = {var, id, document.write},

FIRST(3’’) = FIRST(L) = FIRST(2’) = {var, id, document.write}, FIRST(1) = FIRST(L) = {var, id, document.write}

Lastly we compute the FOLLOW. All FOLLOW sets are initialized to empty except for **FOLLOW(<program>)** = $, where $ means the end of the input.

For L, from rule 1, we put $ in **FOLLOW(L).** For L’, from rule 2’, we put $ in **FOLLOW(L’).** For S, from rule 2’, we put FIRST(L’) = {var, id, document.write} in FOLLOW(S). Moreover, since L’ is nullable, again from rule 2’, we put the current members in FOLLOW(L), i.e. $, in FOLLOW(S). Now we have **FOLLOW(S)** = {var, id, document.write, $}. For S’, from rule 4’, we put the current members of **FOLLOW(S)** in FOLLOW(S’). For <expr>, from rule 5’’, we put current FOLLOW(S’) in FOLLOW(<expr>). From rule 7, we put “)” in **FOLLOW(<expr>),** which is now {var, id, document.write,) }

The second iteration will not change the contents of FOLLOW sets, we reach the following result:

Nonterminals	**FOLLOW**
<program>	$
L	$
L’	$
S	var, id, document.write, $
S’	var, id, document.write, $
<expr>	var, id, document.write, $,)

We can now build the LL(1) parsing table:

	id	num	Document.write	(	)	var	=	$
<program>	1		1			1		
L	2’		2’			2’		
L’	3’’		3’’			3’’		3’
S	6		7			4’		
S’	5’		5’			5’	5’’	5’
<expr>	8	9						

For illustration, we apply the parsing table to input “var a = 1 b = x”. The parsing steps follow left-most derivations. Hence the term LL(1):

<program> → L → S L’ → var id S’ L’ → var id = <expr> L’ → var id = num L’ → var id = num L → var id = num S L’ → var id = num id = <expr> L’ → var id = num id = id L’ → var id = num id = id

In the above, the left most nonterminal is the one on top of the parsing stack. The terminals to its left are popped off the stack by a series of match(token) actions.

Example 2:

1. P → E
2. E → E – T
3. E → T
4. T → T * F
5. T → F
6. F → id
7. F → (E)
8. F → num

This grammar contains left-recursion, which makes the FIRST sets of some right-hand sides (for the same nonterminal) overlap. We must remove left recursion. This is in general a complex task, but we only require simple ones that deal with the immediately obvious left recursion as in this example.

To remove the left recursion for E, we change rules 2 and 3 to the following:

2’. E → T E’
3’. E’→ - T E’
3’’ E’ → ε

The reason we make the above changes is because we know E -> … -> T – T – T … - T or E → T
Our form of right recursion (for E’) is different from the more intuitive for of E → T – E (for E). We use the new form to make sure that semantic actions can later be performed with correct associativity observed. The old intuitive form may easily violate the left associativity when used for propagating semantic values, as we demonstrated before.

Similarly, we change rules 4 and 5 to:

4’. T -> F T’
5’. T’ → * F T’
5’’ T’ → ε

In the following, we omit the steps for computing the FIRST sets. Instead, we illustrate the computation of FOLLOW sets because it is a more complicated task.

Initialize FOLLOW(P) to {$}. **FOLLOW(E')** = FOLLOW(E), **FOLLOW(T')** = FOLLOW(T). From rule 1, we put $ in FOLLOW(E), and from rule 7, we put ")" in FOLLOW(E).
To compute FOLLOW(T) we need to consider both FIRST(E') and FOLLOW(E) because of rule 2' and because E' is nullable. Hence we have **FOLLOW(T**) that contains "-", $, and ")". Similarly, from rule 4', we know FOLLOW(F) contains both FIRST(T') and FOLLOW(T), thus **FOLLOW(F)** = {*,) , -, $}.

We now have the following LL(1) parsing table:

	Id	Num	(	)	-	*	$
P	1	1	1				
E	2'	2'	2'				
E'				3''	3'		3''
T	4'	4'	4'				
T'				5''	5''	5'	5''
F	6	8	7				

Applying to input "3 – 2 – x", we have the following derivation sequence

P → E → T E' → F T' E' → num T' E' → num E' → num – T E' → num – F T' E' → num – num T' E' → num – num * F T' E' → num – num * id T' E' → num – num * id E' → num – num * id

Again, the left most nonterminal in each step is the top-of-stack symbol after matching top of stack terminals to the input.

3. Recursive – Descend Parsing

In class, we illustrate the difficulty of semantic value propagation with LL(1) table-driven parsing. The main problem is the potentially long steps between when a semantic attribute, e.g. the value of a constant (i.e. token num), is available and when that value is needed in a later semantic action, e.g. when the number is used an operand for subtraction. During these steps where to store the semantic value that is easily and reliably retrieved, that is a question. In bottom up parsing, we do not have this difficulty, because the value of num is in the semantic stack until it is time to do reduce. With top-down parsing, the token num is popped off the parsing stack, but the value of num must be kept somewhere.

There exist complicated scheme to manage the semantic stack (which will look very different from the parsing stack) to enable semantic value propagation. It can be quite error prone. Many people prefer to use the recursive-descend approach in which semantic values are easily kept as local variables in the routine that implements a nonterminal before they are needed in later semantic actions.

In recursive-descend parsing, every nonterminal is implemented by a routine. If the nonterminal has multiple rules, then the next token in the input determines which rule to apply next. Each rule is implemented by a branch in a switch statement that processes the right hand side symbols one by one. Each terminal is matched to the next token in the input. Each nonterminal causes its corresponding routine to be invoked.

Semantic values can be propagated as the return value of a nonterminal's routine, e.g. the result of subtraction performed for E'→ - T E' in our example. This is called *synthesized attribute.* The semantic values may also be propagated as parameters of a routine, which are called inherited attributes. For example, in E' → - T E', before the execution of E'(int a) , a holds a number to be subtracted from. Assuming T returns the value of the second operand, a – T. value will be the difference that is be passed to the recursive call to E' for the last nonterminal in the right-hand side, i.e. calling E'(difference).

A pseudo-code skeleton of the translation scheme can then look like the following:

```
Void P() { next_token() = get_token(); printf("%d\n", E()); }   \\ expecting E() to return the final result

Int E() {
        Return( E'( T( ) );
}

Int E'(int a) {
        Switch (next_token) {
        Case '-': match('-'; int  b = a – T(); return E'(b); break;
        Case ')': Case 'eof': return (a); break;
        Otherwise error();
        }
}

Int T() {
        Return( T'( F( ) );
}

Int T'(int a) {
        Switch (next_token) {
        Case '*': match('*'; int  b = a * F(); return T'(b); break;
        Case ')': Case 'eof': Case '-', return (a); break;
        Otherwise error();
        }
}
```

The processing routine for F is omitted.