Introduction to Language Theory and Compilation
Solutions
Session 6: First sets, Follow sets and LL(1) parsing

Solutions

Ex. 1. With regards to the grammar:

1. Give the $\text{First}^1(A)$ and the $\text{Follow}^1(A)$ sets for each $A \in V$.

Symbol	**First**$^1()$	**Follow**$^1()$
S	`begin`	
program	`begin`	$
statement list	`ID read write`	`end`
statement tail	`ID read write` ε	`end`
statement	`ID read write`	`ID read write end`
id list	`ID`	`)`
id tail	`,` ε	`)`
expr list	`( ID INTLIT`	`)`
expr tail	`,` ε	`)`
expression	`( ID INTLIT`	`; , )`
primary tail	`+ -` ε	`; , )`
primary	`( ID INTLIT`	`+ - ; , )`
add op	`+ -`	`( ID INTLIT`

2. Give the First^2(<expression>) and the Follow^2(<expression>) sets.

First$^2(< expression >) = \{$ *((, (ID, (INTLIT, ID+, ID-, INTLIT+, INTLIT-, ID, INTLIT* $\}$

Follow$^2(< expression >) = \{$ *,(, ,ID, ,INTLIT,);,)+,)-,),,)), ;read, ;write, ;ID, ;end* $\}$

Ex. 2. Which grammars are LL(1)?

1. not LL(1) because $a \in \text{Follow}^1(A)$, $a \in \text{First}^1(A)$ and we have the rule $A \to \varepsilon$. Thus, $M[A,a] = \{\text{Produce 2}, \text{Produce 3}\}$.
2. LL(1).
3. LL(1).
4. not LL(1) because $b \in \text{First}^1(A)$, $b \in \text{Follow}^1(B)$ and we have the rule $B \to \varepsilon$. Thus, $M[A,b] = \{\text{Produce 3}, \text{Produce 4}\}$.

Ex. 3. Give the action table of the grammar:

	`-`	`(`	`)`	`ID`	`$`
<S>	1	1		1	
<Expr>	2	3		4	
<ExprTail>	5		6		6
<Var>				7	
<VarTail>	9	8	9		9

Ex. 4. Program a recursive descent parser (in Java, C, C++, ...) for rules (15) through (22).

We start by computing the action table:

	(	)	ID	INTLIT	+	-	,	;
<expression>	15		15	15				
<primary_tail>		17			16	16	17	17
<primary>	18		19	20				
<add_op>					21	22		

Then, each line of the table (ie variable) corresponds to a function which tries to match the next token with the possible symbols and derives the rules accordingly. For instance, when trying to derive <primary>, the parser examines the next token: if it is a `(`, it corresponds to rule (18) <primary> → `(` <expression> `)`, so it first matches the `(`, then derives <expression>, then matches the closing `)`. If it is an ID, it corresponds to rule (19), and it simply matches ID, idem for INTLIT (rule (20)). Finally, empty cells correspond to syntax errors.

The case of <primary_tail> is a bit different, since this variable can be derived into epsilon. In this case, we first check whether the next token belongs to Follow(<primary_tail>) = {`;`, `,`, `)`}, in which case it corresponds to rule (17) <primary_tail> → ε and the function directly returns without doing anything. Otherwise, we should check whether the next token is `+` or `-`, which corresponds to rule (16) and derive accordingly. Note however that in this implementation, we chose to raise syntax errors as low as possible in the parse tree, because they are usually more informative. That is why we actually do not check whether `tok` is `+` or `-` before trying to derive (16) <primary_tail> → <add_op> <primary> <primary_tail>, contrary to what is indicated in the action table (only the `+` and `-` cells display rule (16) in the <primary_tail> line).

```
void expression(){
    primary(); primary_tail();
}
void primary_tail(){
    token tok = next_token();
    // primary_tail is nullable, we check with the follow set
    // and return if primary_tail has to be epsilon
    switch(tok){
    case ';':
    case ',':
    case ')': return;
    }
    // otherwise, we try to match the rule
    add_op(); primary(); primary_tail();
}
void primary(){
    token tok = next_token();
    switch(tok){
    case '(':
        match('('); expression(); match(')');
        break;
    case ID:
        match(ID);
        break;
    case INTLIT:
        match(INTLIT);
        break;
    default:
        syntax_error(tok); break;
    }
}
```

```
void add_op(){
    token tok = next_token();
    switch(tok){
    case '+':
        match('+');
        break;
    case '-':
        match('-');
        break;
    default:
        syntax_error(tok); break;
    }
}
```

Similar to this, you should be able to derive a recursive descent parser for the ALGOL-0 language. Computing the parse-tree while parsing just requires a small modification of the above structure.