HW4 Solution - CS314 Fall24

1 First, Follow, and Predict Sets

(a) First(A) = {b}
First(B) = {b}
First(C) = {c, ε}

(b) Follow(B) = {EOF, c, b}
Follow(C) = {b}

(c) Predict(<C> ::= c<C>) = {c}
Predict(<C> ::= ε) = {b}

2 LL(1) Grammar

(a) If the provided example program has a valid derivation from the grammar, it is correct

Sample Answer:
BEG f (a, b): \n \t a = b + 1 END EOF

(b) Definition:
IFF for any nonterminal with two or more distinct rules:
If the **predict sets** of the **distinct rules** for the **non-terminal** do not share symbols
i.e. PREDICT(<A> ::= α) ∩ PREDICT(<A> ::= β) = ∅
the grammar is indeed LL(1)

LL(1) can be proved by showing the predict sets of non-terminal symbols with multiple production rules, and showing that the sets do not contain the same symbols. This is equivalent to an LL(1) parse table having no more than one rule per cell.
To grade this, please refer to the LL(1) parse table built from Predict sets below. The non-terminals of interest are: <funcname>, <morevars>, <morestmts>, <stmt>, <term>, <variable>, <digit>

Commented [CT1]: -If p is of the form <A> ::= α<B>β, then
-for each such <B>
-if ε ∈ FIRST(β)
-Place {FIRST(β) - ε, FOLLOW(<A>)} in FOLLOW(<B>)

Following this, because ε ∈ First(C), looking at the second rule, you should place FIRST(C) - ε, Follow(A) in Follow(B).

Commented [CT2]: Instead of writing /n and /t terminals, students might (for clarity) write the example program showing the newline and tab reflected in the shape. If this is the case, and there is a clear and obvious discretion for these escape sequences, then full credit should be given

(c) LL(1) Parse Table

To Begin, for each non-terminal, produce the **First** set.

First(program) = {BEG}
First(funcname) = {f, g}
First(arguments) = {(}
First(morevars) = {",", ε}
First(block) = {\t}
First(stmtlist} = {\t}
First(morestmts) = {\n, ε}
First(stmt) = {a, b, c, if, return}
First(assign) = {a, b, c}
First(condition) = {a, b, c}
First(ifstmt) = {if}
First(returnstmt) = {return}
First(expr) = {a, b, c, 0, 1, 2}
First(term) = {a, b, c, 0, 1, 2}
First(variable) = {a, b, c}
First(digit) = {0, 1, 2}

Next, since some First sets contain epsilon, produce the **Follow** sets.

Follow(program) = {END}
Follow(funcname) = {(}
Follow(arguments) = {:}
Follow(morevars) = {)}
Follow(block) = {END}
Follow(stmtlist) = {END}
Follow(morestmts) = {END}
Follow(stmt) = {\n, END}
Follow(assign) = {\n, END}
Follow(condition) = {:}
Follow(ifstmt) = {\n, END}

Commented [CT3]: Comma as a symbol in the set rather than an element delimiter

Follow(returnstmt) = {\n, END}
Follow(expr) = {:, \n, END}
Follow(term) = {:, \n, END, +}
Follow(variable) = {“,”,), =, <=, :, \n, END, +}
Follow(digit) = {:, \n, END, +}

Number the Rules in order of appearance. Extra numbers should be given for OR cases (i.e. multiple rules for one nonterminal)

1) <program> ::= BEG <funcname> <arguments> : \n <block> END EOF
2) <funcname> ::= f
3) | g
4) <arguments> ::= (<variable> <morevars>)
5) <morevars> ::= , <variable> <morevars>
6) | ε
7) <block> ::= <stmtlist>
8) <stmtlist> ::= \t <stmt> <morestmts>
9) <morestmts> ::= \n <stmtlist>
10) | ε
11)<stmt> ::= <assign>
12) | <ifstmt>
13) | <returnstmt>
14)<assign>. ::= <variable> = <expr>
15)<condition> ::= <variable> <= <expr>
16)<ifstmt> ::= if <condition> : <assign> \n \t else : <assign>
17)<returnstmt> ::= return <variable>
18)<expr> ::= <term> + <term>
19)<term>. ::= <variable>
20) | <digit>
21)<variable> ::= a

22) | b
23) | c
24)<digit> :: = 0
25) | 1
26) | 2

From these distinctly numbered rules, let's **build the Predict Set**.
Use First(RHS) – {ε} ∪ **Follow**(LHS) if ε is in the First(RHS).
Otherwise, use **First**(LHS).

Rule	Predict Set
1	BEG
2	f
3	g
4	(
5	“;”
6	)
7	\t
8	\t
9	\n
10	END
11	a,b,c
12	if
13	return
14	a,b,c
15	a,b,c
16	if
17	return
18	a,b,c,0,1,2
19	a,b,c
20	0,1,2
21	a
22	b

23	c
24	0
25	1
26	2

To construct the table, use the PREDICT definition. Notice how for every nonterminal, if there are two or more rules, there is no more than one rule per cell. Indicating that the predict sets of the rules of every nonterminal with multiple rules **do not share symbols** and proves LL(1).

	BEG	f	g	(	,	\t	\n	a	b	c	if	return	0	1	2	)	END
program	1																
funcname		2	3														
arguments				4													
morevars					5											6	
block						7											
stmtlist						8											
morestmts							9										10
stmt								11	11	11	12	13					
assign								14	14	14							
condition								15	15	15							
ifstmt											16						
returnstmt												17					
expr								18	18	18			18	18	18		
term								19	19	19			20	20	20		
variable								21	22	23							
digit													24	25	26		

Commented [CT4]: Example interpretation:
Rule 6 is <morevars> ::= ε.
Because the RHS produces epsilon, take the first of RHS and the follow of LHS and put those symbols in the PREDICT of the LHS. In this case, it's just epsilon, so use Follow of <morevars>.
So, if the LL(1) parser has <morevars> on the stack and sees a) symbol, rule 6 is applied <morevars> is replaced with ε.

Commented [CT5]: Example interpretation:
Rule 11 is <stmt> = <assign>.
Because <assign> cannot produce epsilon, you take symbols from FIRST<assign> as the lookahead, which is a,b,c. So, if the LL(1) parser has stmt on the stack and sees either a, b, or c, rule 11 is applied and stmt gets replaced with assign,

(d) Recursive Descent Parser

The parser should follow the logic that: non-terminals of the grammar are functions that call either **next token and true** if it can product a terminal or **another nonterminal function** if it can produce another nonterminal. It should follos the rules of the LL(1) parse table.

```
main : {
        token = next_token();
        if(program()) print "accept";
        else print "reject";
}
bool program() {
        if(token != "BEG") return false;
        token = next_token();
        if(!funcName()) return false;
        if(!arguments()) return false;
        if(token != ":") return false;
        token = next_token();
        if(token != "\n") return false;
        token = next_token();
        if(!block()) return false;
        if(token != "END") return false;
        token = next_token();
        return true;
}
bool funcname() {
        if(token == "f") {
                token = next_token();
                return true;
        } else if(token == "g") {
                token = next_token();
                return true;
        }
```

```
        return false;
}
bool arguments() {
        if(token != "(") return false;
        token = next_token();
        if(!variable()) return false;
        if(!morevars()) return false;
        if(token != ")") return false;
        token = next_token();
        return true;
}
bool morevars() {
        if(token == ",") {
                if(token != ",") return false;
                token = next_token();
                return (variable() && morevars());
        } else if(token == ")"
                return true;
        } else {
                return false;
        }
}
bool block() {
        return stmtlist();
}
bool stmtlist() {
        if(token != "\t") return false;
        token = next_token();
        return (stmt() && morestmts());
}
bool morestmts() {
        if(token == "\n") {
                token = next_token();
                return stmtlist();
        } else if(token == "END") {
```

```
            return true;
        } else {
            return false;
        }
}
bool stmt() {
        if(token in ["a", "b", "c"]) {
            return assign();
        } else if(token == "if") {
            return ifstmt();
        } else if(token == "return") {
            return returnstmt();
        } else {
            return false;
        }
}
bool assign() {
        if(!variable()) return false;
        if(token != "=") return false;
        token = next_token();
        return expr();
}
bool condition() {
        if(!variable()) return false;
        if(token != "<=") return false;
        token = next_token();
        return expr();
}
bool ifstmt() {
        if(token != "if") return false;
        token = next_token();
        if(!condition()) return false;
        if(token != ":") return false;
        token = next_token();
        if(!assign()) return false;
```

```
        if(token != "\n") return false;
        token = next_token();
        if(token != "\t") return false;
        token = next_token();
        if(token != "else") return false;
        token = next_token();
        if(token != ":") return false;
        token = next_token();
        return assign();
}
bool returnstmt() {
        if(token != "return") return false;
        token = next_token();
        return variable();
}
bool expr() {
        if(!term()) return false;
        if(token != "+") return false;
        token = next_token();
        return term();
}
bool term() {
        if(token in ["a", "b", "c"]) {
                return variable();
        } else if(token in ["0", "1", "2"]) {
                return digit();
        } else {
                return false;
        }
}
bool variable() {
        if(token == "a") {
                token = next_token();
                return true;
        } else if(token == "b") {
```

```
            token = next_token();
            return true;
        } else if(token == "c") {
            token = next_token();
            return true;
        } else {
            return false;
        }
}
bool digit() {
        if(token == "0") {
            token = next_token();
            return true;
        } else if(token == "1") {
            token = next_token();
            return true;
        } else if(token == "2") {
            token = next_token();
            return true;
        } else {
            return false;
        }
}
```