Assignment2

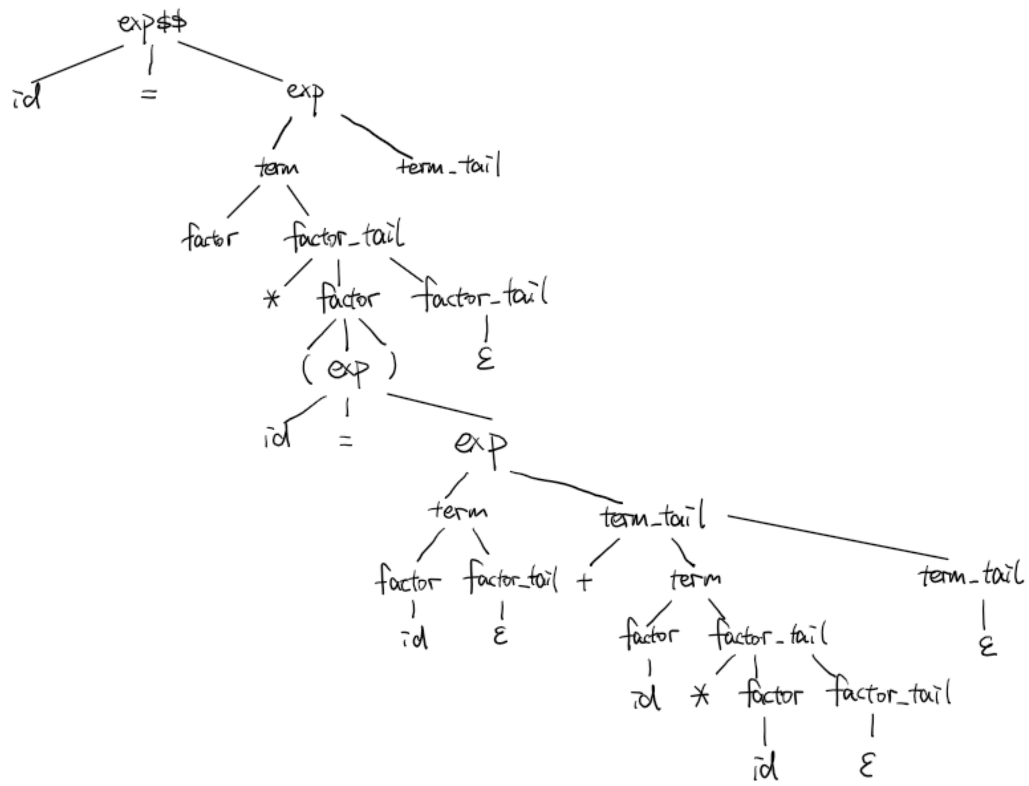
Yaopeng Xie

251195812

1. (30pt) Consider a language where assignments can appear in the same context as expressions; the value of `a = b = c` equals the value of `c`. The following grammar, G, generates such expressions that includes assignments in addition to additions and multiplications:

0.	*program*	$\longrightarrow$	*exp* `$$`
1.	*exp*	$\longrightarrow$	`id` `=` *exp*
2.	*exp*	$\longrightarrow$	*term term_tail*
3.	*term_tail*	$\longrightarrow$	`+` *term term_tail*
4.	*term_tail*	$\longrightarrow$	ε
5.	*term*	$\longrightarrow$	*factor factor_tail*
6.	*factor_tail*	$\longrightarrow$	`*` *factor factor_tail*
7.	*factor_tail*	$\longrightarrow$	ε
8.	*factor*	$\longrightarrow$	`(` *exp* `)`
9.	*factor*	$\longrightarrow$	`id`

(a) (3pt) Show a parse tree for the string: `id = id * (id = id + id * id)$$`.



(b) (10pt) For each production $A \longrightarrow \alpha$, compute $\text{FIRST}(\alpha)$ and $\text{FOLLOW}(A)$ using the algorithm below; $\text{FIRST}(\alpha)$ is computed by `string_FIRST`(α). For each token added, indicate the pair $(step, prod)$ used to add it, where $0 \le step \le 3$ is the step in the algorithm (marked as $\boxed{0}$, $\boxed{1}$, $\boxed{2}$, $\boxed{3}$ below) and $0 \le prod \le 9$ is the production involved; indicate $(0, -)$ when step $\boxed{0}$ is used for terminals.

```
-- EPS values and FIRST sets for all symbols:
    for all terminals c, EPS(c) := false; FIRST(c) := {c} [0]
    for all nonterminals X, EPS(X) := if X ⟶ ϵ then true else false; FIRST(X) := ∅
    repeat
        ⟨outer⟩ for all productions X ⟶ Y1 Y2 ... Yk,
            ⟨inner⟩ for i in 1..k
            [1] add FIRST(Yi) to FIRST(X)
                if not EPS(Yi) (yet) then continue outer loop
            EPS(X) := true
    until no further progress

-- Subroutines for strings, similar to inner loop above:

    function string_EPS(X1 X2 ... Xn)
        for i in 1..n
            if not EPS(Xi) then return false
        return true

    function string_FIRST(X1 X2 ... Xn)
        return_value := ∅
        for i in 1..n
            add FIRST(Xi) to return_value
            if not EPS(Xi) then return

-- FOLLOW sets for all symbols:
    for all symbols X, FOLLOW(X) := ∅
    repeat
        for all productions A ⟶ α B β,
        [2] add string_FIRST(β) to FOLLOW(B)
        for all productions A ⟶ α B
                or A ⟶ α B β, where string_EPS(β) = true,
        [3] add FOLLOW(A) to FOLLOW(B)
    until no further progress

-- PREDICT sets for all productions:
    for all productions A ⟶ α
        PREDICT(A ⟶ α) := string_FIRST(α) ∪ (if string_EPS(α) then FOLLOW(A) else ∅)
```

FIRST(exp$$) = {“id”-(1, 1), “(”-(1, 2), “id”-(1, 2)}

FIRST(id = exp) = {“id”-(0, -)}

FIRST(term term_tail) = {“(”-(1, 5), “id”-(1, 5)}

FIRST(+ term term_tail) = {“+”-(0, -)}

FIRST(ε) = ∅

FIRST(factor factor_tail) = {“(”-(1, 8), “id”-(1, 9)}

FIRST(* factor factor_tail) = {“*”-(0, -)}

FIRST(ε) = ∅

FIRST((exp)) = {“(”-(0, -)}

FIRST(id) = {“id”-(0, -)}

FOLLOW(program) = {∅}

FOLLOW(exp) = {“$$”-(2, 0), “)”-(2, 8)}

FOLLOW(term) = {“+”-(2, 3), $$-(3, 2), “)”-(3, 2)}

FOLLOW(term_tail) = {$$-(3, 2), “)”-(3, 2)}

FOLLOW(factor) = {“*”-(2, 6), $$-(3, 5), “)”-(3, 5)}

FOLLOW(factor_tail) = {$$-(3, 5), “)”-(3, 5)}

(c) (5pt) For each production $i, 1 \le i \le 9$, compute PREDICT(i).

PREDICT(1) = {“id”}

PREDICT(2) = {“(”, “id”}

PREDICT(3) = {“+”}

PREDICT(4) = {$$, “)”}

PREDICT(5) = {“(”, “id”}

PREDICT(6) = {“*”}

PREDICT(7) = {$$, “)”}

PREDICT(8) = {“(”}

PREDICT(9) = {“id”}

(d) (2pt) Using the information computed above, show that this grammar is not LL(1). (See definition on the slide 19 of the LR-parsing chapter.)

there is a conflict in PREDICT: for exp, PREDICT(1) and PREDICT(2) both can get “id”, therefore, this grammar is not LL(1).

(e) (10pt) Modify this grammar to make it LL(1). Explain clearly your changes and prove it is LL(1).

delete program 1: exp -> id = exp

ADD program 10: factor_tail -> = factor factor_tail

Since we added a new program, we need to calculate the PREDICT again.

PREDICT(2) = {“(”, “id”}, PREDICT(3) = {“+”}, PREDICT(4) = {$$, “)”}, PREDICT(5) = {“(”, “id”}, PREDICT(6) = {“*”}, PREDICT(7) = {$$, “)”}, PREDICT(8) = {“(”}, PREDICT(9) = {“id”}, PREDICT(10) = {“=”}.

Now since we delete exp -> id = exp, there will be no conflict on exp, and also there’s no conflict on other terminals. Therefore, after the modification, the grammar is LL(1).

2. (30pt) Consider Boolean expressions containing operands (`id`), operators (`and`, `or`), and parentheses, where `and` has higher precedence than `or`.

(a) (10pt) Write an SLR(1) grammar, G, which is not LL(1), for such expressions, which obeys the precedences indicated.

0. program -> exp$$

1. exp -> exp or term

2. exp -> term

3. term -> term and factor

4. term -> factor

5. factor -> (exp)

6. factor -> id

(b) (5pt) Compute the FIRST(X) and FOLLOW(X) sets for all nonterminals X and PREDICT(i) sets for all productions i.

FIRST(program) = {"(", "id"}

FIRST(exp) = {"(", "id"}

FIRST(term) = {"(", "id"}

FIRST(factor) = {"(", "id"}

FOLLOW(program) = ∅

FOLLOW(exp) = {$$, "or"}

FOLLOW(term) = {$$, "and"}

FOLLOW(factor) = {$$}

PREDICT(0) = {"(", "id"}

PREDICT(1) = {"(", "id"}

PREDICT(2) = {"(", "id"}

PREDICT(3) = {"(", "id"}

PREDICT(4) = {"(", "id"}

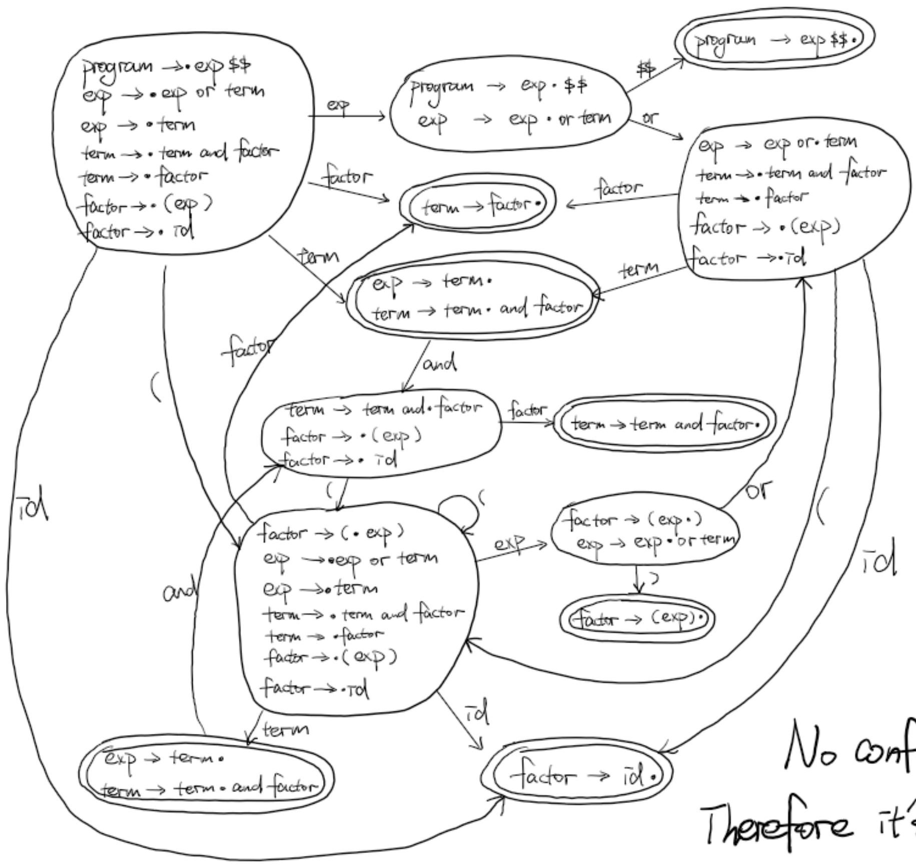
PREDICT(5) = {"("}

PREDICT(6) = {“id”}

(c) (5pt) Prove that G is not LL(1).

There are conflicts in PREDICT, for exp, program 1 and 2 both can get “(” and “id”; for term, program 3 and 4 both can get “(” and “id”.

(d) (10pt) Prove that G is SLR(1) by drawing the SLR graph and show there are no conflicts. Build the graph as shown in the examples we did in class (and done by jflap), not the condensed form in the textbook. For each state with potential conflicts (two LR-items, one with the dot in the middle, one with the dot at the end), explain clearly why there is no shift/reduce conflict.



No conflict

Therefore it's SLR(1)

3. (40pt) Consider the C-style **switch** statement.

(a) (25pt) Write an S-attributed LL(1) grammar that generates C-style **switch** statements and checks that all labels of the arms of the **switch** instructions are distinct. In order to do that, the starting nonterminal, S, will have an attribute **dup** that will store all the duplicate values. There are no duplicate values on the arms of the **switch** statement if and only if $S.\mathtt{dup} = \emptyset$. Therefore, your grammar is required to eventually compute the attribute **dup** of S.

For simplicity, assume that the conditional expression of the **switch** statement and the constant expressions labelling the arms are **expr** tokens and that each arm has a statement that is a **stmt** token; the **break** and **default** parts are omitted as their role is irrelevant for our problem. Each **expr** has an attribute **val** provided by the scanner that gives the value of the expression.

Explain why your grammar works as required. For LL(1), you can use jflap to compute the parse table and show there is no conflict; include the jflap answer (whole window) in your answer.

1. S -> switch (expr) { A }
    - initialize S.dup := ∅ when starting the parse.
    - Initialize seen_value := ∅
2. A -> case expr : stmt A
    - If expr.val in seen_value, then A.dup := A.dup ∪ {expr.val}
    - Else seen_value := seen_value ∪ {expr.val}
    - S.dup := A.dup
3. A -> ε
4. stmt -> ··· (other statements in C)
5. stmt -> ε

Prove LL(1):

PREDICT(1) = switch

PREDICT(2) = case

PREDICT(3) = $$

There is no conflict in PREDICT, therefore the grammar is LL(1)

How the grammar works as required:

The grammar checks for duplicate when generating a switch statement, the nonterminal S starts with an empty set for S.dup and seen_list. Each time a case is processed in program 2, it will check if expr.val has already in seen_value, if it is, then add expr.val to A.dup, else add the expr,val to seen_list and continue. Finally, it adds A.dup to S.dup

(b) (15pt) Using the above attributed grammar, draw a decorated parse tree for the following **switch** instruction:

```
switch ( expr ) {
    case 2 :
    case 3 : stmt
    case 2 : stmt
    case 1 :
    case 2 :
    case 1: stmt
}
```

Show all attributes and arrows indicated what attributes are used to compute each value.