July 24, 2020

CS360 Quiz 3

Group members: Shivanshi Nagar, Stephen Hansen, Dennis George

Start Time 12:40 pm
End Time 2:10 pm

Question 1

In the figure below (of slides 10-15 of Week 3 Part 4 file) we evaluated our expression in normal order. What would happen if we tried to use applicative order? Justify your answer.

$$
\begin{array}{lll}
 & (\underline{\lambda f}.\lambda g.\lambda h.fg(h\,h))\underline{(\lambda x.\lambda y.x)}h(\lambda x.x\,x) & \\
\rightarrow_\beta & (\lambda g.\underline{\lambda h}.(\lambda x.\lambda y.x)g(\underline{h\,h}))h(\lambda x.x\,x) & (1)\\
\rightarrow_\alpha & (\underline{\lambda g}.\lambda k.(\lambda x.\lambda y.x)g(k\,k))\underline{h}(\lambda x.x\,x) & (2)\\
\rightarrow_\beta & (\underline{\lambda k}.(\lambda x.\lambda y.x)h(k\,k))\underline{(\lambda x.x\,x)} & (3)\\
\rightarrow_\beta & (\underline{\lambda x}.\lambda y.x)\underline{h}((\lambda x.x\,x)\ (\lambda x.x\,x)) & (4)\\
\rightarrow_\beta & (\underline{\lambda y}.h)\underline{((\lambda x.x\,x)\ (\lambda x.x\,x))} & (5)\\
\rightarrow_\beta & h & (6)
\end{array}
$$

Normal - Apply the subexpression arguments into the body of the lambda first and only evaluate (reduce) when necessary.
Applicative - Lambda's arguments are fully evaluated (no further reduction can be done) before the arguments are applied to the body of the lambda

Evaluating this expression in applicative order would result in an infinite **β** reduction on line 5 (note that arguments on lines 1-4 are fully evaluated and cannot be reduced any further with applicative order evaluation). Applicative order requires us to evaluate ((λx.x x)(λx.x x)) before we can apply (λy.h) to it. Applying a **β** reduction to ((λx.x x)(λx.x x)) would simplify to exactly the same value: ((λx.x x)(λx.x x)). After this reduction, this value needs to continue to be evaluated due to the parenthesis; the **β** reduction just used can be repeated again. This reduction will continue to occur and we would never get to the step where we would need to apply (λy.h) in order to get the final answer *h*. So under normal order, we get h (as the argument in line 5 is never evaluated) but under applicative order we evaluate the argument of line 5 and run into an infinite loop.

Question 2

The parse table follows the PREDICT set directly. The algorithm used for PREDICT set is,

for all productions A → α
PREDICT(A −→ α) := string FIRST(α) ∪ (if string EPS(α) then FOLLOW(A) else ∅)

The PREDICT set is directly computed from FIRST, FOLLOW and EPS sets.

Top-of-stack nonterminal	Current input token											
	`id`	`number`	`read`	`write`	`:=`	`(`	`)`	`+`	`-`	`*`	`/`	`$$`
program	1	–	1	1	–	–	–	–	–	–	–	1
stmt_list	2	–	2	2	–	–	–	–	–	–	–	3
stmt	4	–	5	6	–	–	–	–	–	–	–	–
expr	7	7	–	–	–	7	–	–	–	–	–	–
term_tail	9	–	9	9	–	–	9	8	8	–	–	9
term	10	10	–	–	–	10	–	–	–	–	–	–
factor_tail	12	–	12	12	–	–	12	12	12	11	11	12
factor	14	15	–	–	–	13	–	–	–	–	–	–
add_op	–	–	–	–	–	–	–	16	17	–	–	–
mult_op	–	–	–	–	–	–	–	–	–	18	19	–

PREDICT

1. *program* ⟶ *stmt_list* `$$` {`id`, `read`, `write`, `$$`}
2. *stmt_list* ⟶ *stmt stmt_list* {`id`, `read`, `write`}
3. *stmt_list* ⟶ ϵ {`$$`}
4. *stmt* ⟶ `id :=` *expr* {`id`}
5. *stmt* ⟶ `read id` {`read`}
6. *stmt* ⟶ `write` *expr* {`write`}
7. *expr* ⟶ *term term_tail* {`(`, `id`, `number`}
8. *term_tail* ⟶ *add_op term term_tail* {`+`, `-`}
9. *term_tail* ⟶ ϵ {`)`, `id`, `read`, `write`, `$$`}
10. *term* ⟶ *factor factor_tail* {`(`, `id`, `number`}
11. *factor_tail* ⟶ *mult_op factor factor_tail* {`*`, `/`}
12. *factor_tail* ⟶ ϵ {`+`, `-`, `)`, `id`, `read`, `write`, `$$`}
13. *factor* ⟶ `(` *expr* `)` {`(`}
14. *factor* ⟶ `id` {`id`}
15. *factor* ⟶ `number` {`number`}
16. *add_op* ⟶ `+` {`+`}
17. *add_op* ⟶ `-` {`-`}
18. *mult_op* ⟶ `*` {`*`}
19. *mult_op* ⟶ `/` {`/`}

The table consists of tokens as columns and nonterminals as rows.
We start with using algorithms to make our FIRST and FOLLOW sets. FIRST(A) is the set of all tokens that could be the start of an A (set of all possible tokens that could be the first token

produced by A) and FOLLOW(A) is the set of all tokens that could come after an A in some valid program (set of all tokens that could follow the production of A). Once there are no options left (no other cases, nothing to add to FIRST and FOLLOW) we construct the PREDICT set.
For each production rule, we link the ‘non terminals’ with the ‘input symbols’ using the production number. A dash indicates a syntax error.
Example:

- The first production, with non-terminal ‘program’ will have id, read, write and $$ (the symbol columns) as the tokens in its “predict” set.
- The 3rd production, with non-terminal ‘stmt_list’ will have $$ as the token in its “predict” set.
- The 19th production, with non-terminal ‘mult_op’ will have / as the token in its “predict” set.

The predict set is critical as it allows a program, given a current symbol at the top of the stack and an incoming symbol, which production to use to evaluate and parse the expression. If a symbol belongs to more than one prediction of the productions of a nonterminal, then it is impossible for the compiler to predict which production to use (not a LL(1) grammar). The table can only be constructed if each cell corresponding to a nonterminal and an input symbol has a unique prediction.

Question 3

S -> (S)S | epsilon

Note: using $ to denote end of input.

FIRST(S) = { (, epsilon }
FOLLOW(S) = {), $ }
PREDICT(S -> (S)S) = { (}
PREDICT(S -> epsilon) = {), $ }

This grammar is LL(1) because each token belongs to a unique PREDICT set for each production of S. If a token were to belong to more than one production with the same left-hand side, then the grammar would not be LL(1) as the parser would not be able to choose which production to employ when the left-hand side is at the top of the parse stack and we see the token coming up in the input. LL(1) denotes a left-to-right leftmost derivation with 1 token of lookahead. If a token was in more than one prediction set for a left-hand nonterminal, then it would not be LL(1) as this single token of lookahead would not allow the parser to make a deterministic choice for the production rule to apply (since the token belongs to multiple sets).

Since left parenthesis only belongs to PREDICT of production S -> (S)S and right parenthesis and end-of-input ($) belong to PREDICT of production S -> epsilon, a LL(1) parser would always be able to choose a single production to employ for each token. Each predicted token maps to a unique production for S.

Question 4

Grammar iii of figure 23 of week 1-2:

E -> TE’
E’ -> +TE’ | epsilon
T -> FT’
T’ -> *FT’ | epsilon
F -> (E) | id

FIRST(E) = { (, id }
FIRST(E’) = { +, epsilon }
FIRST(T) = { (, id }
FIRST(T’) = { *, epsilon }
FIRST(F) = { (, id }
FOLLOW(E) = {), $ }
FOLLOW(E’) = {), $ }
FOLLOW(T) = { +,), $ }
FOLLOW(T’) = { +,), $ }
FOLLOW(F) = { *, +,), $ }
PREDICT(E -> TE’) = { (, id }
PREDICT(E’ -> +TE’) = { + }
PREDICT(E’ -> epsilon) = {), $ }
PREDICT(T -> FT’) = { (, id }
PREDICT(T’ -> *FT’) = { * }
PREDICT(T’ -> epsilon) = { +,), $ }
PREDICT(F -> (E)) = { (}
PREDICT(F -> id } = { id }

This grammar is LL(1) because each token belongs to a unique PREDICT set, unique production per each left-hand side. If a token were to belong to more than one production with the same left-hand side, then the grammar would not be LL(1) as the parser would not be able to choose which production to employ when the left-hand side is at the top of the parse stack and we see the token coming up in the input. LL(1) denotes a left-to-right leftmost derivation with 1 token of lookahead. If a token was in more than one prediction set for a left-hand nonterminal, then it would not be LL(1) as this single token of lookahead would not allow the parser to make a deterministic choice for the production rule to apply (since the token belongs to multiple sets).

In this problem for E, the only tokens that production E -> TE’ could predict are (and id. Similarly for E’, no tokens overlap between either of its predicted productions, and the same goes for the other variables. Although there is some token overlap between productions in general, none of these overlaps are within productions with the same left hand side, so there is never any ambiguity when predicting a production given a current symbol at the top of the stack and an incoming token. Hence this grammar is LL(1).