15-411: Compiler Design — Spring 2024

Recitation 3: Lexing and Parsing Solutions — 9 February

Lexing

Checkpoint 0

Remember that the lexer is responsible for reading in an input program/string and producing a stream of tokens/symbols that are then later consumed by the parser. As an exercise, try lexing the following segment of a C0 program. You may choose whatever textual representation you deem best for each symbol (i.e "(" $\rightarrow$ "LPAREN").

```
if (score < 100) {
    return 1;
}
```

Solution:

if	(	score	<	100	)	{	return	1	;	}
IF	LPAREN	IDENT	LESSTHAN	DECCONST	RPAREN	LBRACE	RETURN	DECCONST	SEMI	RBRACE

Grammars & Parsing

Now once you have a stream of tokens from the lexer, the parser can now construct a parse tree from the stream of tokens. Recall from lecture a grammar G for a language L(G) is defined by a set of productions $\alpha \to \beta$ and a start symbol S, a distinguished non-terminal symbol.

For a given grammar G with start symbol S, a derivation in G is a sequence of rewritings $S \to \gamma_1 \to \to \gamma_n = w \in L(G)$ in which we apply productions from G. **Parsing** uses this derivation process to produce a parse tree (derivation) for w, in which the nodes represent the non-terminal symbols and the root being S.

We run into ambiguities when there are multiple possible parse trees for the same token stream. Below we'll take a closer look at possible ambiguities.

Grammar Ambiguities

Ambiguities can result as a consequence of the production rules and symbols chosen in defining a grammar G. An ambiguity in the grammar arises when there are multiple possible valid parse trees for the same token stream.

Checkpoint 1

Given the context-free grammar G containing productions of the form:

$$\begin{array}{llcl} \gamma_1 & : & A & \to & \mathbf{A + A} \\ \gamma_2 & : & A & \to & \mathbf{A \text{ - } A} \\ \gamma_3 & : & A & \to & \text{int} \\ \gamma_4 & : & A & \to & \text{id} \end{array}$$

Prove that the grammar G is ambiguous by showing two parse trees for the stream $1 + 2 - \mathtt{id_x}$.

Solution:
Parse Tree 1: $A \to A + A \to 1 + (A - A) \to 1 + (2 - A) \to 1 + (2 - id_x)$
Parse Tree 2: $A \to A - A \to (A + A) - A \to (1 + 2) - A \to (1 + 2) - id_x$

Conflicts in a LR(k) Parser

Now we will discuss shift-reduce and reduce-reduce conflicts common in LR(k) parsers. Remember from lecture that a bottom-up LR(k) parser parses from left-to-right in a single-pass with right-most derivation using k look-ahead tokens. A shift-reduce parser holds viable prefixes on a stack along with `k` lookahead symbols with the input stream containing remaining symbols.

LR(k) parsers at each step must determine whether the parser should *shift* or *reduce*. *Shifting* saves the current token on the maintained stack and reads another token while *reducing* applies some rule from the grammar to the front of the current token stack. As such, LR(k) parsers are prone to two common issues when dealing with certain grammars: **shift-reduce** and **reduce-reduce** conflicts.

Shift-Reduce Conflicts

A shift-reduce conflict occurs when it is ambiguous whether the parser should *shift* or *reduce*.

Checkpoint 2

Show that the following grammar has a shift-reduce conflict by showing two different ways to parse the string `200 * 2 + 11`.

Then, explain how you would resolve this conflict.

$E \to E + E$

$E \to E * E$

$E \to [0-9]*$

$E \to (E)$

Solution:

```
          || 200 * 2 + 11
      200 || * 2 + 11
        E || * 2 + 11
      E * || 2 + 11
    E * 2 || + 11
    E * E || + 11

(Reduce E*E to E)      or   (Shift '+')
E || + 11              or   E * E + || 11
  ...                                ...
```

There are multiple ways to resolve this conflict, but a simple one is to assign a higher precedence to the multiplication operator.

Reduce-Reduce Conflicts

A reduce-reduce conflict occurs when more than one rule in the grammar can be applied.

Checkpoint 3

Show that the following grammar has a reduce-reduce conflict by showing a successful and an unsuccessful parse of the string `bbbc`.

Then, explain how you would resolve this conflict.

$S \to Cc$

$S \to Dd$

$C \to \epsilon$

$C \to Cb$

$D \to \epsilon$

$D \to Db$

Solution:

```
         || bbbc
 C || bbbc   or   D || bbbc
Cb || bbc        Db || bbc
 C || bbc         D || bbc
Cb || bc         Db || bc
 C || bc          D || bc
Cb || c          Db || c
 C || c           D || c
Cc ||            Dc ||
 S ||               ??
```

There are multiple ways to resolve this conflict, but in a LR(1) parser, one must modify the grammar by getting rid of one of the conflicting productions. Using a parser with arbitrary lookahead would also solve this issue.