Recitation 4: Lexing and Parsing Solutions 13 February

Solution:

[TA only: Remember to Announce]

- L2 Compiler due **tomorrow** Saturday 2/14.

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 "(" → "LPAREN").

```
1   if (score < 100) {
2      return 1;
3   }
```

Solution:

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

IF  LPAREN  IDENT  LESSTHAN  DECCONST  RPAREN  LBRACE  RETURN  DECCONST  SEMI  RBRACE
```

Grammars & Parsing

Recall from lecture that 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 nonterminal symbol. Then a derivation in G is a sequence of rewritings $S \to \gamma_1 \to \cdots \to \gamma_n = w$ in which we repeatedly apply productions from G until we obtain a string w consisting entirely of terminal symbols. The language $L(G)$ is the set of all such w's that can be derived in this manner.

Context-free grammars restrict the LHS α of productions to be a single nonterminal symbol A. They are called context-free because A can always be replaced by β regardless of its context, i.e. what symbols surround it. The subset of C0 that you parse for labs 1-3 can be described by a context-free grammar. **Context-sensitive** grammars allow α to be more than a single nonterminal symbol. Productions can take the form $\alpha A \beta \to \alpha \gamma \beta$ where α and β are the context of, or strings surrounding A, and they can determine whether A may be replaced with γ. The full grammar of C0 is context-sensitive, as will the labs starting with lab 4.

Parsing is the process which receives a stream of tokens (typically from the lexer) representing w, and figures out the derivation in G which could produce w. The derivation identified is known as a *parse tree* in which the root is S, the internal nodes are various nonterminal symbols, and the leaves are terminal symbols.

Grammar Ambiguities

Ambiguities can result from 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:

$$\begin{array}{lcll} \gamma_1 & : & A & \rightarrow & \textbf{A + A} \\ \gamma_2 & : & A & \rightarrow & \textbf{A - A} \\ \gamma_3 & : & A & \rightarrow & \text{int[n]} \\ \gamma_4 & : & A & \rightarrow & \text{id} \end{array}$$

Prove that the grammar G is ambiguous by showing two parse trees for the stream $1 + 2 - \texttt{id}_\texttt{x}$.

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

Conflicts in a Shift-Reduce LR(k) Parser

A bottom-up LR(k) parser parses from **l**eft-to-right in a single-pass, produces a **r**ightmost derivation in reverse, and uses **k** look-ahead tokens. It is a shift-reduce parser, which holds viable prefixes on a stack, along with the input stream containing the remaining symbols. At each step, it must determine whether to *shift* or *reduce*.

- *Shifting* consumes the next token of the input stream and pushes it to the stack.
- *Reducing* applies some rule from the grammar to replace the top symbols of the current stack with a nonterminal that can produce them.

To help decide what steps to take, the parser may use the lookahead, or peek at up to *k* future tokens of the input stream. The goal is to proceed until the stack contains only the start symbol S and the input stream is fully consumed. Afterward, the full parse tree can be built by examining the sequence of shift/reduce steps.

Grammar ambiguities, discussed in the previous checkpoint, can cause a LR(k) parser to get stuck. This happens in two ways: **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.

$$\begin{array}{lcll} \gamma_1 & : & E & \rightarrow & \textbf{E + E} \\ \gamma_2 & : & E & \rightarrow & \textbf{E * E} \\ \gamma_3 & : & E & \rightarrow & \text{int[n]} \\ \gamma_4 & : & E & \rightarrow & \textbf{(E)} \end{array}$$

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. A fancier solution is to change the grammar rules to look something like

γ_1 : E $\rightarrow$ **E + F**
γ_2 : E $\rightarrow$ **F**
γ_3 : F $\rightarrow$ **F * A**
γ_4 : F $\rightarrow$ **A**
γ_5 : A $\rightarrow$ int[n]
γ_6 : A $\rightarrow$ (**E**)

This new grammar uses stratification to ensure that multiplication is parsed with higher precedence than addition. A multiplication subexpression is parsed as F, which can then be added together to form an E. Integers and parenthesized expressions are parsed even more tightly as A, which can be multiplied together to form an F.

In addition to the conflict in this checkpoint, the old grammar also suffers from the issue described in checkpoint 1, which is that $+$ and $*$ can ambiguously associate in any order. This new grammar enforces left associativity, so that an expression like $3 + 4 + 5$ is always parsed as $(3 + 4) + 5$.

Reduce-Reduce Conflicts

A reduce-reduce conflict occurs when more than one production 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.

γ_1 : S $\rightarrow$ **C**c
γ_2 : S $\rightarrow$ **D**d
γ_3 : C $\rightarrow$ ϵ
γ_4 : C $\rightarrow$ **C**b
γ_5 : D $\rightarrow$ ϵ
γ_6 : D $\rightarrow$ **D**b

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. solve this issue.
Using a parser with arbitrary lookahead would also