Lemon (Objectives)

- Given an LALR(1) grammar, the student will be able to write a Lemon specification for the grammar.

1

Lemon

- Lemon is a parser generator.
  - Similar to Bison and Yacc
- It takes a restricted form of a context-free grammar, G, and produces C code that recognizes strings in L(G).
- We will discuss how lemon works in this class.

2

Lemon File Layout

- Free format
  - Grammar rules and declarations can be at any point of the input file
  - However, I suggest
    - Declarations first
    - Followed by grammar rules

3

Terminals and Non-terminals

- A terminal (token) is any string of alphanumeric and underscore characters that begins with an upper case letters
  - Convention: make terminals all upper case
- A non-terminal, on the other hand, is any string of alphanumeric and underscore characters than begins with a lower case letter.
  - Again, the usual convention is to make non-terminals use all lower case letters.
- Terminal and non-terminal symbols do not need to be declared

4

Precedence Rules

- %left, %right and %nonassoc
  - declare the associativity of tokens
  - lexical order in file gives precedence
  - all tokens on the same line have the same precedence

```
%left PLUS MINUS.
%left TIMES DIVIDE.
%right EXP.
```

  - PLUS and MINUS have lowest precedence, EXP has the highest

Type

- %type nt {C type}
  - declare a type for the attributes associated with non-terminals
  - allows you to pass information up the parse tree as rules are reduced
  - may have different type of information for different grammar symbols
- %token_type {C type}
  - Default is int if not specified
  - Same type for all tokens

Lemon Directives

- %start_symbol
  - declare the start symbol of the grammar
    - %start_symbol prog
  - By default, the first non-terminal in the grammar file
- %include {}
  - Specify C code that will be included in the beginning of the generated file
- %name
  - Specify the prefix of generated functions

Lemon Productions

- Form

```
lhs     ::=  rhs.    {//code for action}
```

- Example

```
expr    ::= expr PLUS expr.     {// generate an ADD inst}

expr    ::=  ID.                {// load an var in a reg}
```

Attributes

- Attributes allow you to pass information up the tree as reductions are performed.

```
expr(e)    ::= expr(rhs1) PLUS expr(rhs2).
                { e = generateAddInstruction(rhs1, rhs2);}
expr(e)    ::= ID (id).
                { e = generateLoad(id);}
```

  - The %type declaration for a symbol tells Lemon which type to use for a particular attribute
  - In the scanner, assign initial attribute of terminals and send to the parser

9

Example – Communication Flex/Lemon

From parser file

```
%token_type {char *}
%type program {char *}
%start_symbol program

program  ::= ID (id).
             {
               printf("Program Name: %s\n", id);
             }
```

From scanner file

```
/* directly assigning yytext to tokenAttr
      may lose the info later. */
[a-zA-Z]([a-zA-Z0-9])*  { tokenAttr =
                            yytext;
                          return(ID);
                        }
```

10

Calling the Parser

- To invoke the parser to process the next token, call the function

  Parse(parser, token_id, token_attribute)

  - Note that %name MYNAME will replace Parse by MYNAME in all parser functions
- Initialize the parser with

```
void* parser;

parser = ParseAlloc(malloc);
```

11