Homework 2 Questions

Top
debugging parser conflicts
disambiguating grammars

YACC/BISON IN PRACTICE
Getting Started with Bison

- [https://www.gnu.org/software/bison/](https://www.gnu.org/software/bison/)
- [https://sourceforge.net/projects/gnuwin32/](https://sourceforge.net/projects/gnuwin32/) for Windows (but Cygwin or WSL works better)

- Run `yacc -d` or `bison -yd` `<name>.y` to get two files:
  - `<name>.tab.c` implements the parser
  - `<name>.tab.h` defines tokens and values for use in lexing – include in `<name>.lex`

- How does it know whether the grammar is LR(1)? It doesn’t!
  - When run, reports number of shift/reduce and reduce/reduce conflicts
  - `yacc -dv` or `bison -ydv` also produces `<name>.output`, which describes the parser states and conflicts
Precedence and Associativity Declarations

- Parser generators like yacc/bison often support precedence and associativity declarations
  - Resolve common cases of shift/reduce conflicts without refactoring

  ```
  %left PLUS
  %left TIMES
  ```

- Tokens can be declared `left`, `right`, or `nonassoc`
- Tokens further down have higher precedence (bind tighter, get evaluated first)
- Precedence of a `rule` is based on the precedence of its `last` terminal:
  
  ```
  E \rightarrow E + E  
  1 + 2 . * 3  
  E \rightarrow E * E  
  ```

- Can’t apply precedence to nonterminals
Example: Ambiguity in Real Languages

• Consider this grammar:
  \[ S \rightarrow \text{if } (E) \; S \]
  \[ S \rightarrow \text{if } (E) \; S \; \text{else } S \]
  \[ S \rightarrow X = E \]
  \[ E \rightarrow \ldots \]

• Consider how to parse:
  \[ \text{if } (E_1) \; \text{if } (E_2) \; S_1 \; \text{else } S_2 \]

• This is known as the “dangling else” problem.

• What should the “right” answer be?

• How do we change the grammar?
How to Disambiguate if-then-else

• Want to rule out:

\[
\text{if (E}_1\text{)}\begin{cases} 
\text{if (E}_2\text{) } S_1 \\
\text{else } S_2 
\end{cases}
\]

• Rule: An un-matched ‘if’ should not appear as the ‘then’ clause of a containing ‘if’.

\[
\begin{align*}
S & \mapsto M \mid U & \quad & \text{ // } M = \text{“matched”}, \ U = \text{“unmatched”} \\
U & \mapsto \text{if (E) } S & \quad & \text{ // Unmatched ‘if’} \\
U & \mapsto \text{if (E) } M \text{ else } U & \quad & \text{ // Nested if is matched} \\
M & \mapsto \text{if (E) } M \text{ else } M & \quad & \text{ // Matched ‘if’} \\
M & \mapsto X = E & \quad & \text{ // Other statements}
\end{align*}
\]
Alternative: Use {} 

- Ambiguity arises because the ‘then’ branch is not well bracketed:

```plaintext
if (E₁) { if (E₂) { S₁ } } else S₂  // unambiguous
if (E₁) { if (E₂) { S₁ } else S₂ }  // unambiguous
```

- So: could just require brackets
  - But requiring them for the else clause too leads to ugly code for chained if-statements:

```plaintext
if (c₁) {
    ...
} else {
    if (c₂) {
        ...
    } else {
        if (c₃) {
            ...
        } else {
            ...
        }
    }
}
```

So, compromise? Allow unbracketed else block only if the body is ‘if’:

```plaintext
if (c₁) {
    ...
} else if (c₂) {
    ...
} else if (c₃) {
    ...
} else {
    ...
}
```

Benefits:
- Less ambiguous
- Easy to parse
- Enforces good style
Questions

Top
First and Follow Sets

- First($S\$) = First($S$) = First($ES'$) = First($E$)
- First(number) = { number }
- First( ( $S$ ) ) = { ( }
- So First($S\$) = First ($ES'$) = { number, ( }
- First($\varepsilon$) = ? (not applicable)
- First(+ $S$) = { + }

Follow($S'$) = Follow($S$)
Follow($S$) = { $, ) }

<table>
<thead>
<tr>
<th></th>
<th>number</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$ (EOF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$\rightarrow S$</td>
<td></td>
<td>$\rightarrow S$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>$\rightarrow ES'$</td>
<td></td>
<td>$\rightarrow ES'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S'$</td>
<td></td>
<td>$\rightarrow S$</td>
<td>$\rightarrow \varepsilon$</td>
<td>$\rightarrow \varepsilon$</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>$\rightarrow$ num.</td>
<td></td>
<td>$\rightarrow ( S )$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$T \rightarrow S\$
$S \rightarrow ES'$
$S' \rightarrow \varepsilon$
$S' \rightarrow + S$
$E \rightarrow$ number $| ( S )$
First and Follow Sets

• First($T$) = First($S$) = First($E$) = { number, ( } 
• First($S'$) = { + } 

• Follow($S'$) = Follow($S$) 
• Follow($S$) = { $, ) } 

<table>
<thead>
<tr>
<th></th>
<th>number</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$ (EOF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>$\rightarrow S$</td>
<td></td>
<td></td>
<td>$\rightarrow S$</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>$\rightarrow ES'$</td>
<td></td>
<td></td>
<td>$\rightarrow ES'$</td>
<td></td>
</tr>
<tr>
<td>$S'$</td>
<td></td>
<td>$\rightarrow S$</td>
<td></td>
<td>$\rightarrow \varepsilon$</td>
<td>$\rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$E$</td>
<td>$\rightarrow \text{num.}$</td>
<td></td>
<td>$\rightarrow ( S )$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Questions

Top
Abstract Syntax Trees

Source Code
(Character stream)
if (b == 0) { a = 1; }

Token stream:
if ( b == 0 ) { a = 0 ; }

Abstract Syntax Tree:
```
  If
    Eq
      b
    Assn
      0
      a
      1
    None
```

Assembly Code
```
11:
  cmpq %eax, $0
  jeq 12
  jmp 13
12:
  ...
```
Abstract Syntax Trees in Code

• One type per nonterminal
• Enum indicating which production was used
• Union for associated data and child nodes

```
struct A_exp { // corresponds to E
    enum { E_num, E_parens } kind;
    union { int val; A_stm child } u;
}

struct A_stm { // corresponds to S
    enum { S_plus, S_single } kind;
    union { struct { A_stm left; A_exp right; } addition; A_exp e; } u;
}
```

```
S ⟷ S + E | E
E ⟷ number | ( S )
```
• Write a constructor for each kind of AST node

```
S ⟷ S + E | E
E ⟷ number | ( S )
```

```c
struct A_exp {  // corresponds to E
    enum { E_num, E_parens } kind;
    union { int val; A_stm child } u;
};

A_exp A_NumNode(int val){
    A_exp result;
    result.kind = E_num;
    result.u.val = val;
}
```
Abstract Syntax Trees in Code

• Write a constructor for each kind of AST node

\[ S \rightarrow S + E \mid E \]
\[ E \rightarrow \text{number} \mid (S) \]

```c
struct A_exp {    // corresponds to E
    enum { E_num, E_parens } kind;
    union { int val; A_stm child } u;
};

A_exp A_NumNode(int val){
    A_exp result;
    result.kind = E_num;
    result.u.val = val;
}

A_exp A_ParenNode(A_stm s){
    A_exp result;
    result.kind = E_parens;
    result.u.child = s;
}
```
Abstract Syntax Trees in Code

- Write a *constructor* for each kind of AST node

\[
S \rightarrow S + E \mid E \\
E \rightarrow \text{number} \mid (S)
\]

- Then use the constructors in actions:

```c
exp:
   LPAREN stm RPAREN { return A_ParenNode($2); }
```

```c
A_exp A_NumNode(int val){
   A_exp result;
   result.kind = E_num;
   result.u.val = val;
}
```

```c
A_exp A_ParenNode(A_stm s){
   A_exp result;
   result.kind = E_parens;
   result.u.child = s;
}
```
Goal: take the sequence of tokens produced by lexing, and turn them into an AST, to make the structure of the program clear

Method:
1. Write a file (.y, .grm) that describes the grammar and disambiguates it, and explains how to build the AST node for each production
2. Use a parser generator to turn that grammar into a shift-reduce automaton, encoded in an action-goto table

Lexer generators and parser generators work together to define the syntax of a language

Lexing and parsing together turn program text into a representation that starts to suggest meaning/behavior of programs

Next up: looking for meaning