Parsing

COMP360

The "Alan Turing Law" is an informal term for the law in the United Kingdom (received Royal Assent on January 31, 2017), which serves as an amnesty law to pardon men who were cautioned or convicted under historical legislation that outlawed homosexual acts. The provision is named after Alan Turing, the World War II codebreaker and computing pioneer, who was convicted for gross indecency in 1952. Turing received a royal pardon (posthumously) in 2013.

Wikipedia

Homework

- Two assignments are posted on Blackboard

1. Answer 9 questions that include FSA and Mealy machine parse tables
   - Due by noon today
2. Write a parser for the Snowflake language
   - Due by noon on Friday, February 10, 2017

Parsing

- Parsers input a list of tokens from the lexical scanner, check the syntax of the input program and generate a parse tree for later code generation
- The syntax of the language being parsed is defined by a BNF

General Parsing Algorithms

- Any CFG can be parsed in $O(n^3)$ time and $O(n^2)$ for unambiguous grammars
- There are well-known parsing algorithms that permit this
  - Early algorithm
  - Cooke-Younger-Kasami (CYK) algorithm
- An $O(n^3)$ time parser becomes too slow for large input source

Parsing Methods

- Top Down Parsing
  - Traces a leftmost derivation
  - Builds a parse tree in preorder
  - LL and Recursive Descent are top down
- Bottom Up Parsing
  - Builds a parse tree starting at the leaves
  - Produces the reverse of a rightmost derivation
  - LR and LR-K algorithms are bottom up

Parsing classes

- There are large classes of grammars for which we can build parsers that run in linear time
- **LL** stands for **L**eft-to-right, **L**eftmost derivation
  - LL parsers are top down
- **LR** stands for **L**eft-to-right, **R**ightmost derivation
  - LR parsers are bottom up

Looking Ahead

- You commonly see LL or LR written with a number in parentheses after it
- This number indicates how many tokens of look-ahead are required in order to parse
- Almost all real compilers use one token of look-ahead

from *Programming Language Pragmatics* by Scott

Fortran Compiling

- The Fortran language allows spaces in the middle of a variable, which can contain letters and numbers

```
DO 5 J = 1,35      // for loop
DO 5 J = 1.35      // assignment statement
```

- This makes Fortran more difficult to parse
- The parser needs a long look ahead in this case

Equivalence

- Every LL(1) grammar is also LR(1), though right recursion in production tends to require very deep stacks and complicates semantic analysis
- Every CFL that can be parsed deterministically has an SLR(1) grammar (which is LR(1))
- Every deterministic CFL with the *prefix property* (no valid string is a prefix of another valid string) has an LR(0) grammar

A Context Free Language can be recognized by

A. Regular Expression
B. Finite State Automaton
C. Deterministic Finite Automaton
D. Push down automaton
E. All the above

Parse Tables

- Both LL and LR parse algorithms use parse tables
- Decisions are based on the current state, the input symbol and the top of stack symbol
- For each input symbol, the algorithm determines what steps to take based on the parse table

Scanning Tools

- Lex is a computer program that generates lexical analyzers
- Lex is commonly used with the yacc parser generator
- Lex was created by Mike Lesk and Eric Schmidt in 1975
- Lex is guided by a file that associates regular expression patterns with C statements
- Flex (fast lexical analyzer generator) is a free and open-source software alternative to lex

Parsing Tools

- **Yacc** (**Y**et **A**nother **C**ompiler **C**ompiler) is a Look Ahead Left-to-Right (LALR) parser generator
- Developed in the early 1970s by Stephen C. Johnson
- GNU **Bison** is a parser generator
- Flex and Bison are often used together

Recursive Descent

- Recursive descent is a top down parsing method
- It is relatively easy algorithm to implement
- Functions or methods are written to represent each BNF production
- Recursive descent does not use parse tables

BNF to Recursive Descent

- Each production becomes a separate method
- Usually the method name matches the name of the non-terminal
- For each non-terminal on the right side, the method for the non-terminal is called
- For terminal symbols, the program checks if the next token is the proper symbol

Input function

- The input to the parser is a list of tokens from the lexical scanner
- There is an input function (called lex() in the example) that returns the next token
- The current input character is stored in a global variable (called nextToken in the example)

- *This may not be the best way to implement this, but it seemed to be the easiest to understand*

lex() and nextToken

- Frequently the program looks at nextToken to check the next token
- If the token is what is desired, lex() is called to remove that token from the input
- The returned value of lex is sometimes not used in the example because the program knows what it is from the previous examination of nextToken

Example Grammar

expr → term

| term **+** expr

| term **-** expr

term → factor

| factor ***** factor

| factor **/** factor

factor → **variable**

| **(** expr **)**

expr function

expr → term | term + expr | term - expr

```
void expr() {
 term();
 if (nextToken == '+'
        || nextToken == '-') {
    lex();
    expr();
 }
}
```

term function

term → factor | factor * factor | factor / factor

```
void term() {
  factor();
  if (nextToken == '*' ||
          nextToken == '/') {
     lex();
     factor();
  }
}
```

factor → variable | (exp)

```
static void factor() {
  if ( nextToken == variable) {
      lex();
  } else if (nextToken == '(') {
      lex();
      expr();
      if (nextToken == ')') {
            lex();
      } else {
            print "Syntax Error";
      }
  } else {
      print "Syntax Error";
  }
}
```

Main Procedure

```
main() {
    expr();    // start symbol
}
```

CFG can be recognized by a FSA with a stack

Where is the stack in the recursive descent algorithm?

A. No stack is needed
B. The algorithm uses the method call stack
C. An array is used as a stack
D. All the above
E. None of the above

Return Value

- The example recursive descent methods were void
- Alternatively, they may be Boolean methods that return true if the current input tokens match this form and false if they do not

Multiple Possible Paths

- Consider the production

  widget → thing | doodad

- The recursive descent method could be

```
boolean widget() {
    if (thing()) return true;
    return doodad();
}
```

Write a recursive descent method for

crow → (owl hawk)

Possible Solution

crow → (owl hawk)

```
boolean crow() {
        if (nextToken != “(“ ) return false;
        lex();
        if ( !owl() )    return false;
        if ( !hawk() ) return false;
        if (nextToken != “)“ ) return false;
        lex();
        return true;
}
```

Undo

- The recursive descent methods “consume” input tokens as they are executed
- Each method looks at the first token after what was matched by earlier methods
- If a method “fails”, it must leave the tokens on the input in the same position they were at the start of the method
- A rollback method may be necessary

Not All Grammars Are Created Equal

- There is an infinite number of grammars for every context-free language
- Some grammars make parsing easier
- Chomsky Normal Form BNF are in the format:

  A → B C

  A → **x**

- Greibach Normal Form BNF are in the format:

  A → **x**

  A → **x** B

  A → **x** B more terminals and nonterminals

Left Recursion

- Left Recursion occurs when a production defining a non-terminal starts with the non-terminal

  thing → thing + widget

- Indirect Left Recursion has the same effect as direct recursion, but involves multiple rules

  thing → dodad **+** widget

  dodad → thing **x**

Recursion Problems

- The recursive descent algorithm will get stuck with a left recursive rule

```
void thing() {
   thing();          // infinite loop here
   if (nextToken != "+") Syntax error
   widget();
}
```

Right Recursion Correction

- The problems with left recursion can be corrected with right recursion

$$\text{thing} \rightarrow \text{thing} + \text{widget}$$

- can be changed to

$$\text{thing} \rightarrow \text{widget} + \text{thing}$$

Left Recursion Example

exp → exp + term
| term

term → term * factor
| factor

factor → (exp)
| varnum

Corrected BNF

exp → term + exp
| term

term → factor * term
| factor

factor → **(** exp **)**
| varnum

Unique First Terminal

- It is difficult to parse productions that have multiple options that start with the same symbol

```
thing → x widget
      | x dodad
```

- List recursion is generally not a problem

```
A → B | B , A
```

Unique Symbol Grammar

- The previous example can be corrected by adding another non-terminal

```
thing → x whatever
whatever → widget | dodad
```

Homework

- Two assignments are posted on Blackboard

1. Answer 9 questions that include FSA and Mealy machine parse tables
   - Due by noon today
2. Write a parser for the Snowflake language
   - Due by noon on Friday, February 10, 2017