, 12, 13, 14, 15, 16, 17

CS 241 - Foundations of Sequential Programs **Spring 2017**

Lecture 11, 12, 13, 14, 15, 16, 17: June 8th - June 29th, 2017

Lecturer: Kevin Lanctot *Notes By: Harsh Mistry*

11.1 Context-Free Grammars

Example 11.1 *Grammar for strings starting with a's and ending with b's*

- $S \to aSb$
- $S \to \epsilon$

Definition 11.2
- ***Language of a CFG*** *: The set of all valid strings (Sequences of terminals) that can be derived from the start symbol (S)*
- ***G*** *is a context-free grammar*
- ***L(G)*** *is the language specified by G*
- *a* ***derivation*** *: starting with the start symbol, applying a sequence of rules until there are no more non-terminals*
- ***Production Rules*** *capture Union, Concatenation, Recursion*

Context-free grammars consist of a four-tuple `{N,T,P,S}`

- N is a finite set of non-terminals
- T is a finite set of terminals
- P is a finite set of production rules in the form A $\to \beta$ where
- S is the start symbol,

Derivations

- Leftmost Derivation - Always expand the leftmost non-terminal first
- Rightmost Derivation - Always expand the rightmost non-terminal first.

11.1.1 Parse Trees

Parse Trees (Also called derivations trees) visualize the entire derivation at once. The root of the tree is the start symbol, the children are given by subsequent rules, and the leafs are the terminals

Parse trees are processed **Post order** with a depth first traversal.

- Depth first - visit your first child and all its descendants before visiting your second child.
- Post order - process all your children before you do any processing yourself

11.2 Top-Down Parsing

Parsing giving a grammar and a word. involves fining the derivation of w. The two strategies for parsing are :

- Top-down : Find a non-terminal (e.g S) and replace it with the right-hand side
- Bottom-up : Replace a right-hand side with a non-terminal

Stack-Based Parsing

For top-down parsing, we use a stack to remember information about derivations or processed input

Definition 11.3 *The start symbol occurs as the LHS of exactly one rule and that rule must begin and end with a terminal*

Parsing Algorithms

1. To start, push the start symbol, S', on the stack
2. When a non-terminal is at the top of the stack :
   - expand the non-terminal using a production rule where the RHS of the rule matches the input (e.g. if the rule is $S' \rightarrow \vdash S \dashv$ then pop S off the stack and push $\vdash S \dashv$ onto the stack)
3. When it is a terminal at the top of the stack : match with input
   - Pop terminal
   - read the next character from the input

LL(1) Parsing

LL(1) means processing the input from Left to right while finding the Leftmost derivation and looking ahead by 1 token.

To achieve this we implement a Predict table and consider three functions.

- First()
  Determine for each non-terminal what are the possible terminal they can derive on the leftmost side of the string
- Follow()
  For all rules of the form $A \to \beta$ whenever $Empty(\beta)$
- Empty() or Nullable()

Non-LL(1) Grammars

A Non-LL Grammar Is a grammar that is ambiguous and requires us to look ahead to the second symbol in order to tell which rule to use.

11.3 Bottom-up Parsing

- the derivation progresses from the bottom of the parse tree up to the top (i.e. S), i.e. a bottom-up derivation
- current step in derivation: stack + input to be read -
- stack is read from bottom to top

LR Parsing

LR parsing is when input is read normally but is analysed from the right.

There are two operations in LR parsing

- Shift
  - move a character from the input file to the stack
  - well also include it in the Read column to keep track of what has been read so far.
- Reduce
  - If there is a production rule of the form S $\to$ AyB and AyB is on the stack then reduce (i.e. replace) AyB to S
  - this step is the act of applying a production rule to simplify what is on the stack

Building an LR(0) Automation

- Start state: make teh start state the first rule with a dot ($\cdot$) in front of the left most symbol of the RHS
- For each state create a transition out of teh state with the symbol "$\cdot$"
- For non-terminals : If "$\cdot$" precedes a non-terminal, add all productions with that non-terminal on the LHS to the current state (and place the "$\cdot$" in the leftmost position of those rules)

Using an LR(0) Automation

```
for each input token
  read the stack (from the bottom up) + the current input
  do the action indicated for the current input
    if there s a transition out of current state with current input
      then shift (push) that input onto the stack
    if current state has only one item and the rightmost symbol is    .
      then we know we can reduce i.e. {
        pop the RHS of the stack,
        reread the stack (from the bottom-up),
        follow the transition for the LHS,
        push the LHS onto the stack
      }
if S is on the stack when all input is read
  then ACCEPT
```

LR Parsing Limitations

The time complexity is $O(n^2)$, as for each of the n input chars, we move through the stack and automaton up to n times.

To resolve this and to achieve $O(n)$ complexity we store the automation state in the States stack. So if you pop i elements off the Symbol Stack and then pop i elements off the States Stack. It tells you what state to go to next

11.3.1 SLR(1) Parser

- When we add one character of lookahead, we have an SLR(1) (Simple LR with 1 character lookahead) parser
- We modify our existing LR(0) automaton as follows
- When you are in a state that has a rule of the form A $\rightarrow \alpha \cdot \{X\}$ (which calls for a reduction) if the next symbol is X reduce using that rule, otherwise shift.
- Allowing for lookahead and improving time complexity by using a state stack, we now have the following algorithm, where
- the top of the state stack, `state_stack.top`, is the current state
- $\delta(s, t)$ is the new state the automaton goes to when it is in state s and it processes the next token, t

SLR(1) Parsing Algorithm

```
push start_symbol onto the symbol_stack
push state(q0, start) onto the state_stack
  for each token a in the input {
    while (there is a reduction A to g * {a} in state_stack.top ) {
    pop |g| symbols off the symbol_stack // reduce
    pop |g| symbols off the state_stack
    push A on the symbol_stack
    push stat(state_stack.top, A) onto the state_stack
}
  push a onto the symbol_stack // shift
  if (state(state_stack.top, a) == undefined) report parse error
```

```
12    else push state(state_stack.top, a) onto the state_stack
13  }
14  if (end has been shifted)
15  then ACCEPT
```

11.4 Context-Sensitive Analysis

Context-Sensitive Analysis involves analysing code for errors that go beyond Syntax issues such as Undeclared Variables, Duplicate Procedures, invalid types, and out-of-scope variables.

Solving Variable Declaration Issues

To resolve Variable declaration issues we can use a symbol table like a MIPS assembler, but keep track of types of variables as well.

- When using a variable, make sure its is in the symbol table
- When declaring a variable
  - Check that is not already in the symbol table
  - If it it not, then add it
  - If it is, throw an error

For Variable scopes, implement a global symbol table for procedure names and types and have pointers to separate symbol tables for each procedure to track its parameters and local variables.

Type Checking

To type check decorate the parse tree with types and ensure all the rules are followed.

- $\frac{}{\text{NUM} : \text{int}}$
- $\frac{}{\text{NULL} : \text{int*}}$
- $\frac{E{:}\tau}{(E){:}\tau}$
- $\frac{\text{E: int}}{\text{\& E} : \text{int*}}$
- $\frac{\text{E: int*}}{\text{E} : \text{int*}}$
- $\frac{\text{E: int}}{\text{new int[E]} : \text{int*}}$
- $\frac{E_1{:}\text{int}, E_2{:}\text{int}}{E_1 * E_2{:}\text{int}}$
- $\frac{E_1{:}\text{int}, E_2{:}\text{int}}{E_1 / E_2{:}\text{int}}$
- $\frac{E_1{:}\text{int}, E_2{:}\text{int}}{E_1 \% E_2{:}\text{int}}$
- $\frac{E_1{:}\text{int}, E_2{:}\text{int}}{E_1 + E_2{:}\text{int}}$

- $\frac{E_1:\text{int}*, E_2:\text{int}}{E_1+E_2:\text{int}*}$
- $\frac{E_1:\text{int}, E_2:\text{int}*}{E_1+E_2:\text{int}*}$
- $\frac{E_1:\text{int}, E_2:\text{int}}{E_1-E_2:\text{int}}$
- $\frac{E_1:\text{int}*, E_2:\text{int}}{E_1-E_2:\text{int}*}$
- $\frac{E_1:\text{int}*, E_2:\text{int}*}{E_1-E_2:\text{int}}$
- $\frac{E_1:\tau, E_2:\tau}{well-typed(E_1==E_2)}$
- $\frac{E_1:\tau, E_2:\tau}{well-typed(E_1<E_2)}$
- $\frac{E_1:\tau, E_2:\tau}{well-typed(E_1=E_2)}$
- $\frac{E_1:\tau}{well-typed(delete[]E)}$
- $\frac{well-typed(T), well-typed(S_1)}{well-typed(while(T)\{S_1\})}$
- $\frac{well-typed(T), well-typed(S_1), well-typed(S_2)}{well-typed(if(T)\{S_1\}else\{s_2\})}$