Ji Kang

Professor Jim Hogg

CSS 448

9 Feb. 2020

Answer the following questions in the context of a Bottom-Up Parser:

1 **Why is a Bottom-Up parse so called?**

a. Given the program (The leaf nodes of a AST), a Bottom-Up parser works its way upwards, filling in the derivations (production rules) along the way until it reaches the root node.

2 **What is the input to, and the output from, a Bottom-Up parser?**

a. Taking in the grammar of the language as well as a sentence (a program's source code for example), constructs a parse tree by working its way up to the root node.

3 **What happens on a *shift* operation?**

a. We move the current input onto the stack, essentially "shifting" our cursor over one place. The stack represents the **frontier** which is the collection of all terminal and non-terminal symbols the parser has seen so far. The parser will shift because there is no substring of symbols in the frontier to reduce using a production rule **or** that there is a specific reduction the parser wants to do.

4 **What happens on a *reduce* operation?**

a. Looking at elements at the top of the stack, the parser will see if a valid **handle** exists. If so, the reduction is carried out by matching the handle with the right-hand side of a production rule and the left-hand side non-terminal value is pushed onto the stack (after popping the handle values).

5 **What is a *handle*?**

a. A **handle** is a substring of the frontier (all the elements on the stack) that is either length 0 up to the entire length of the frontier. Informally, it is the substring of terminal and non-terminal symbols that match the right-hand side of a valid production rule in the grammar.

6 **What is a *frontier*?**

a. All the symbols the parser has seen so far. In class, we use this cursor analogy. The frontier would be all the symbols on the left of the cursor. Equally, it is also all of the elements on the stack.

7 **What's the difference between a *sentential form*, and a *sentence*?**

a. A **sentential form** is the current string the parser has "decoded up to" so far. It contains a mix of non-terminal and terminal symbols that is on its way to becoming a sentence. A intermediate form if you will.

b. **Sentence** is a string that contains only terminal values. It is the lowest form of the input for it does not contain any non-terminals that can be reduced down using a production rule.

8 **Why is a Bottom-Up parser also called an "LR" parser?**

a. Bottom-Up parsers follow a left-to-right (The **L**) and rightmost derivation (The **R**) scheme. It scans the sentence and builds up to the root node (**Bottom-Up**) and along the way. It's the combination of these two things that a Bottom-Up parser is also called a **LR parser**.

9 **What's the difference between an LR(1) parser and an LR(2) parser?**

a. The parameterized number represent how many tokens ahead the parser is looking at. The only difference between a LR(1) and a LR(2) parser is that an LR(1) is looking one token ahead in its input stream while a LR(2) parser is looking two tokens ahead.

10 **Every AST node in our Tog parser contains the fields: AST kind and Ast* next. What do these two fields contain? Give an example.**

a. "kind" tells us what kind of AST node it is E.g. It could be an expression – ASTEXP, an "if" – ASTIF, a "var" – ASTVAR, and so on.
b. "next" is pointing to the next node in the AST. E.g. If there are multiple functions in the program, a ASTFUN node's "next" field could point to the next function.

11 **The next code snippet shows an AstLet node. Why do we make *every* AST node contain the two fields kind and next?**

```
// Let => "let" Nam "=" Exp
typedef struct {
  AST     kind;       // ASTLET
  Ast*    next;
  char*   nam;        // eg: abc
  AstExp* exp;        // eg: x + 1
} AstLet;
```

a. The AST is a collection of various nodes but there's no strong way to see what "type" it is (E.g. does the node represent a number, expression, parameter, keyword, etc.). This is why "kind" is always needed for all AST nodes.
b. "next" allows chaining of specific nodes to refer to each other. Like the function example I mentioned in problem 10.
c. Not all "next" fields are going to be populated with a non-null value but is needed to allow dimension to the program by allowing multiple functions for example. For nodes in a tree to connect to each other, all nodes start off as generic AST nodes. Kind helps differentiate each node to their actual type. Since some AST_types utilize the "next" field, and all nodes are initially a generic AST node, all nodes essentially have this field as well (though it may not be used at all).

12 What’s the difference between static and dynamic typing?

a. Statically typed values are declared to a hold a value of a certain type at runtime. This can be primitive values such as int, double, float, or long. Dynamic typing is more fluid in the sense, that even though a value may have started as an **integer**, it can be type casted to a **float**, or even a **string** during run-time.

13 Here is a snippet of code, written in some computer language. If this snippet is valid, then is the language statically, or dynamically typed?

```
x = 4
print x
x = “hello”
print x
```

The language is dynamically typed. The value **x** is initially an integer value of 4 but is later changed to a string value of “**hello**”. Memory management during run-time was required to allow this code to run in the sense that space for “x = 4” was initially made but the space had to be deallocated and new space had to be allocated to store “hello” into the variable **x**.