

CS160 – Final
Winter 2008

1 Shift Reduce Parsing

Consider the following grammar

$$\begin{array}{l} A \rightarrow B \\ \quad | \quad z \\ B \rightarrow [B] \\ \quad | \quad B + B \\ \quad | \quad y \end{array}$$

- i. Draw the LR(0) Canonical Collection (the DFA) for the grammar

2 Procedures

- i. Show the contents of a lexically scoped symbol table for the program on the right. (the language is similar to C but with the addition of nested procedures)

```
procedure main(int argc, char* argv)
{
    float x;
    int y;
    procedure P1(int x)
    {
        char j;
        procedure P2(char y)
        {
            char z;
            ...
        }
        ...
    }
    procedure P3(char y)
    {
        char z;
        int a[100];
        int* b = malloc(100,4);
        ...
    }
    ...
}
```

- ii. You observe the following sequence of events. Main starts, call to P1, call to P2, return, return, call to P3, call to P3, return, call to P3, return, return. Draw the activation tree for this sequence of events.

- iii.** Consider the following sequence of events (which is simply truncated from part ii). Main starts, call to P1, call to P2, return, return, call to P3, call to P3. At this point you freeze the machine in the debugger so you can look through the activation records that are on the stack. How many bytes on the stack are storing local variable data (including arrays)? Assume char is 1 byte and that int, float, and pointers are all 4 bytes. State any other assumptions that you need to make, and *show your work*.

3 Min and Max values

Programming a tiny microcontroller can be a serious pain. One tricky thing is that in C the number of bits used to store an integer is dependent on the architecture. For example on x86 platforms integers are 32 bit values (which can store unsigned integers in the range of 0 to 2^{32}), while other architectures store them in 64 bit values (where unsigned integers are in the range of 0 to 2^{64}). Imagine what a headache it is to program an 16-bit microcontroller that can only store the values 0 through 65535. You are going to write an attribute grammar to help make that easier.

Consider the following grammar that describes expressions on unsigned integers (while the grammar is ambiguous you can assume that precedence is enforced correctly during the parsing). You must add two attributes to this grammar, *min* and *max*, which when calculated will produce the minimum and maximum value that a given sub-expression could ever have. If *min* ever goes below 0, set *min* to **error**. If *max* ever goes above 65535, then set *max* to **error**. If *S.min* = **error** or *S.max* = **error** then we know that somewhere in the code there is a problem. If *S.min* is not an error then *S.min* should be the minimum possible value that the full expression could evaluate to. Likewise if *S.max* is not an error then *S.max* should be the maximum possible value that full expression could evaluate to. To simplify things, **you can assume that all variables (var) have a min of 10 and a max of 20 by default**. You can also assume that all constants (num) have an attribute *val* that contains the value of that constant. Below are a couple of example expressions and the results that we should see calculated if the attribute grammar is written correctly

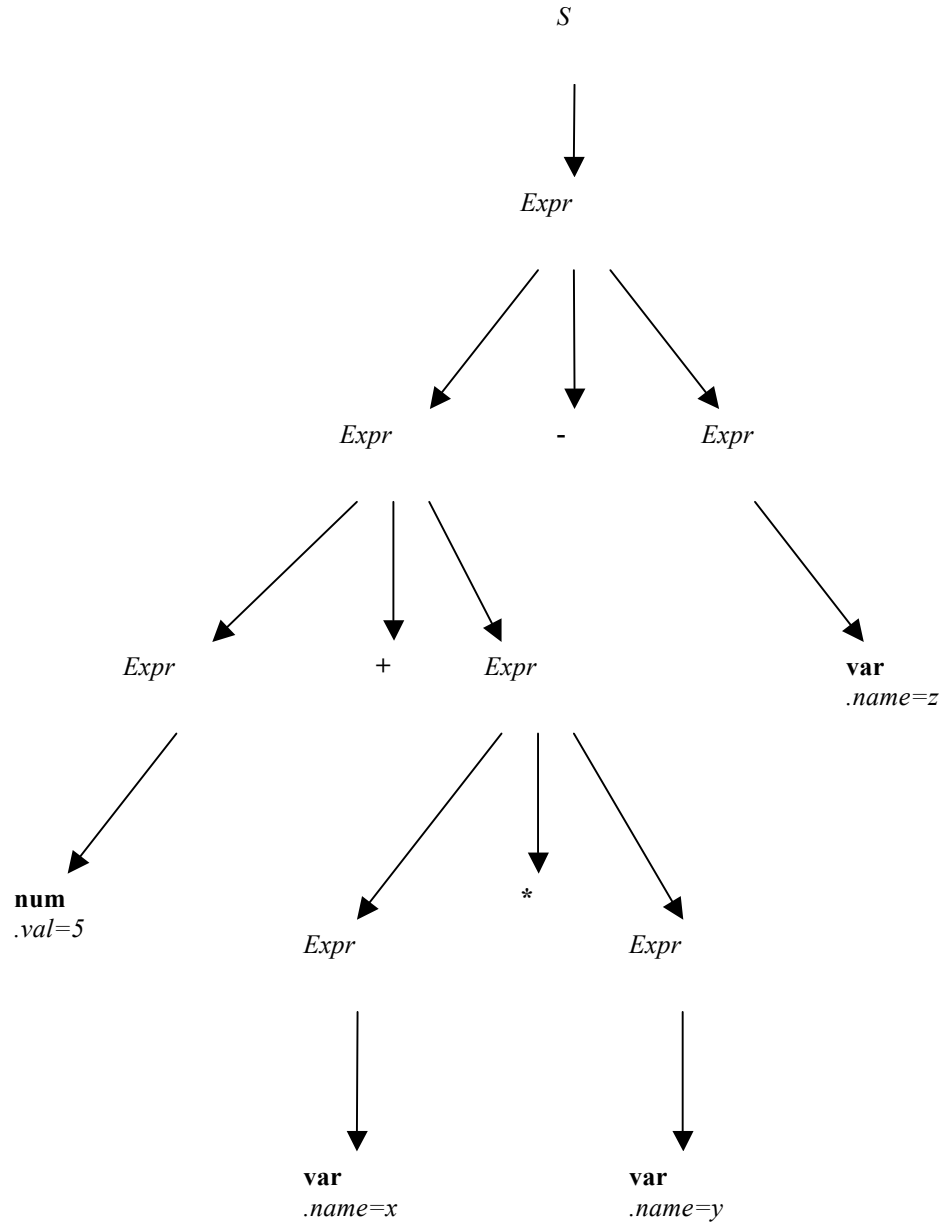
| Example sentence: | Expected Result: | Comments: |
|-------------------|-----------------------|--|
| x | S.min=10, S.max=20 | The minimum value of x is defined above |
| x+5 | S.min=15, S.max=25 | The minimum value of x was 10, but +5 gives 15 |
| x-15 | S.min=error, S.max=5 | The min value is below 0 so it is an error |
| x-15+y | S.min=error, S.max=25 | The min value dipped below 0 before y was added |
| x+y | S.min=20, S.max=40 | This requires thinking about the min and max way way that two variables can be combined together |

| | | |
|---------------------------------|---|---|
| $S \rightarrow Expr$ | { | } |
| $Expr \rightarrow \mathbf{num}$ | { | } |
| var | { | } |
| (Expr) | { | } |
| Expr + Expr | { | } |
| Expr - Expr | { | } |
| Expr * Expr | { | } |

- i. Extend the above grammar (in the space between the brackets) into an attribute grammar that computes *min* and *max* (they should be synthesized).

(problem 4 continued)

- ii. For the following parse tree, show the value of min and max for each node in the tree as it would be calculated using your grammar (i.e. decorate the tree with min and max). The parse tree is for the sentence: **5 + x * y - z**



4 True or False

True False questions. note: **there will be a half-point penalty for incorrect answers** so don't just try and guess on this question.

- a) **T F** The scanner takes $O(n)$ time, where n is the number of characters in the program
- b) **T F** If a grammar is left recursive it is not LR(1)
- c) **T F** If a grammar is left recursive it is not LL(1)
- d) **T F** If a grammar is right recursive it is not LR(1)
- e) **T F** If a grammar is right recursive it is not LL(1)
- f) **T F** Activation Records are allocated at compile time
- g) **T F** The symbol table is used at compile time
- h) **T F** The scanner is built from a state machine
- i) **T F** The scanner can recognize any regular language
- j) **T F** Synthesized attribute grammars are those that are computed from the top-down
- k) **T F** To compute an inherited attribute the attributes of the parents cannot be used
- l) **T F** Predictive top down parsing can only handle LL grammars

5 Arrow Notation

$$\begin{array}{lcl} S & \rightarrow & \textit{funkyexpr} \\ \textit{funkyexpr} & \rightarrow & \textit{funkyexpr} \uparrow \textit{funkyexpr} \\ & | & \textit{funkyexpr} \downarrow \textit{funkyexpr} \\ & | & \textit{funkyexpr} \downarrow \textit{funkyexpr} \\ & | & \mathbf{x} \end{array}$$

Some people *love* notation, and being the compiler people we get to deal with it. Consider the operator grammar above (feel free to abbreviate *funkyexpr* as *f*). We would like to modify the grammar such that the precedence of operations is handled correctly. The operator \downarrow should have the highest precedence, while \downarrow should have the lowest. Furthermore, \uparrow and \downarrow should both be right associative, while \downarrow is left associative.

- i. Modify the grammar above such that the grammar is unambiguous, and that precedence and associativity are handled correctly.

- ii. Show that your modified grammar from part i works by showing the parse tree on the sentence: $x \uparrow x \downarrow x \downarrow x$

7 First, Follow, and LL(1)

Consider the following language:

$$\begin{array}{lcl} S & \rightarrow & A C z \\ A & \rightarrow & y B \\ & | & A x \\ & | & \epsilon \\ B & \rightarrow & w B w \\ & | & \epsilon \\ C & \rightarrow & v \\ & | & \epsilon \end{array}$$

i. For the above grammar, fill out the below first and follow sets

$$\text{FIRST}(S) = \{$$

$$\text{FIRST}(A) = \{$$

$$\text{FIRST}(B) = \{$$

$$\text{FIRST}(C) = \{$$

$$\text{FOLLOW}(S) = \{$$

$$\text{FOLLOW}(A) = \{$$

$$\text{FOLLOW}(B) = \{$$

$$\text{FOLLOW}(C) = \{$$

ii. Is the grammar above LL(1)? Why or why not (justify your answer)

8 First, Follow, and Recursive Decent

Consider the following language:

$$\begin{array}{l} S \rightarrow [S A] \\ \quad | \quad \mathbf{x} \\ A \rightarrow + S B \\ \quad | \quad B \mathbf{y} \\ \quad | \quad \varepsilon \\ B \rightarrow - S A \mathbf{z} \\ \quad | \quad \varepsilon \end{array}$$

- i. Write the function $A()$ for a recursive decent parser for the language shown above. Please be as specific as possible about the lookaheads (i.e. the “default” case should be an error).

- ii. Even though you have not completed Draw the **activation tree** for your recursive decent parser, if it is called on the input “[x – x z y]”. If we pause the execution of your parser right after it finishes matching the second “x”, what will be on the stack and in what order.

9 Abstract Syntax Trees

```
Prog  → VectorExpr
VectorExpr → VectorExpr x VectorExpr // cross product
           | VectorExpr * VectorExpr // scalar product
           | VectorExpr • VectorExpr // dot product
           | Vector
           | num
Vector  → list( num )
```

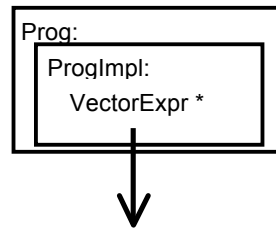
The grammar above is the Abstract Syntax for a language of vector expressions (feel free to abbreviate *VectorExpr* as *VE* and *Vector* as *V*). If we were to build a compiler for this little language we would need a data structure for the Abstract Syntax Tree, and this could be done using the idea of classes and inheritance in the same way that AST class from our projects did, allowing for easy composition and traversal. This question deals with the relationship between an input string (the program to be compiled), the theoretical grammar above, and the shape of the data structures we used in this course.

You may assume that compiler has already handled the ambiguity in this grammar by specifying the precedence and associativity of the operators. The order of precedence, from highest to lowest is $*$, \bullet , \times . Dot product should be right associative while cross product and scalar product should be left associative. In particular, let's consider what our compiler will do with the program:

5 5 5 \times 1 2 3 \bullet 7

- i. Draw a picture of the resulting abstract syntax tree *without* worrying about the underlying data structure (i.e. show me the “shape” of the tree when you account for the precedence and associativity of the operators, but don't worry about the data structure stuff just yet)

- ii. Assume that we use classes and inheritance to build our AST data structure in a way that is completely analogous to the way it was done in the project (without worrying about the whole visitor/visitee thing). Please sketch the resulting data structure for the same sentence you used from part i.



- iii. Consider now that this grammar will still let you do things that don't make sense. In particular, A major problem with the above grammar is that scalars and vectors are treated the same, even though they should not be. A dot product requires two vectors and returns a scalar, a cross product requires two vectors and returns a vector, and a scalar product requires one vector and one scalar (in either order) and returns a vector. Describe what code you would have to write if you wanted to walk over the tree and type check this property. You do not have to actually write the code, just list the steps you would take.

10 Three Address Code Generation

| | |
|-------------------------------|---|
| <code>r1 := r2 + r3</code> | Adds registers r2 and r3 and puts the result in r1 (assume subtraction, multiplication, and division are also included) |
| <code>iflt r1 r2 label</code> | If <code>r1 < r2</code> then jump to label (assume <code>ifgt</code> , <code>iflteq</code> , and <code>ifgteq</code> for <code>></code> , <code><=</code> , and <code>>=</code> respectively are also included) |
| <code>ifbool r1 label</code> | If <code>r1 == 1</code> then jump to label, if <code>r1 == 0</code> then do nothing (fall through), and if r1 is neither 0 nor 1 then it is an error |
| <code>jump label</code> | jump (unconditional) to label |

For this problem we will be dealing with a new type of loop (similar to a do/while loop) called a until-then loop. Consider the following production that defines the syntax of the until-then loop.

$UntilThenStmt \rightarrow \mathbf{do} (Stmt_1) \mathbf{until} Expr \mathbf{then} Stmt_2$

UntilThenStmt, *Expr*, and *Stmt* are nonterminal symbols and **do** and **until** are terminal symbols. Assume that both *Stmt* and *Expr* have a *code* attribute holding the three-address code generated for it. The idea behind an until-then loop is that *Stmt*₁ is executed repeatedly until the expression evaluates to true. When that happens, then *Stmt*₂ is executed. To be clear,

“do (`x=x*2; y++`) until `x>100` then `y--`” would be equivalent to the C while loop

“`x=x*2; y++; while(!(x>100)) {x=x*2; y++;} y--;`”

Your job is to perform the code generation for the *UntilThenStmt*.

- i. Begin by drawing a block diagram of your approach (so I can see what you are trying to do) using the example statement above. It is okay to simplify for clarity and this point, I want to see your high level ideas about how to implement it.
- ii. Write the pseudo-code (either the semantic rules or something closer to the implementation from our project – whichever you are more comfortable with) to generate code for the *UntilThenStmt*. I don't care about your syntax here, show me the order in which the statements must be generated.