Section Notes on Syntax-Directed Translation

October 1, 2003

Recall from lecture that syntax-directed translation allows us to interleave useful computation with parsing, for example performing simple typechecking of the input program as we go. In PA3, you will be using syntax-directed translation to create a recursive-descent parser that constructs ASTs for its input programs. This section will cover again how this translation will work, using the following simple grammar as a running example:

```
E -> E + T | T
T -> T * F | F
F -> int | (E)
```

Recall that in syntax-directed translation, we maintain a *semantic stack* along with the parse stack. The basic rules governing the semantic stack are as follows:

- When a terminal is parsed, its *attribute* is pushed on the stack. For constructing ASTs, the token itself suffices as the attribute for terminals.
- When a non-terminal is parsed, its translation is pushed on the stack. In our case, the translation of a non-terminal will be an AST node.

At the completion of a successful parse, the semantic stack should have one element, the translation of the start symbol. When doing bottom-up (LR) parsing, maintaining the semantic stack is straightforward; we essentially "mirror" the transformations done to the parse stack. When we shift a terminal, we push its attribute on the semantic stack. When we reduce to a non-terminal, we pop the items on the semantic stack corresponding to the items popped from the parse stack, and then push the computed translation for the non-terminal on the semantic stack. See the lecture notes for an example.

Unfortunately, when doing top-down parsing (LL(1) or recursive-descent), performing syntax-directed translation is not so easy, since the translation is performed bottom-up. We solve this problem by adding *actions* to our grammar, specifying how translation is to be performed for each production. In LL(1) parsing, these actions (more specifically, numbers representing the actions) are pushed on the parse stack along with the corresponding production, and then executed when popped. These actions are responsible for popping all the items on the semantic stack corresponding to terminals and non-terminals on the right-hand side of the production, and then pushing the computed translation. An example action for the `E -> E + T` production would be code like the following:

```
ASTNode right = pop();
pop(); // get rid of the plus token
ASTNode left = pop();
push(new InfixExpression(left,right));
```

Other actions are similar. Assigning these actions numbers (as in the lecture notes), we end up with the following grammar (I have left out actions that just push and pop the top element on the semantic stack with no computation):

```
E -> E + T #1 | T
T -> T * F #2 | F
F -> int #3 | (E) #4
```

Unfortunately, this grammar still isn't suitable for top-down parsing since it is left-recursive; we will want to eliminate this left-recursion and left-factor the grammar (see WA3 for why we want to left-factor even with a recursive-descent parser). Luckily, we can play a neat trick while performing this transformation to maintain our actions; just move them around along with everything else! In our example, the left-factored grammar with actions is as follows (we use `_` for ϵ, as in PA3):

```
E -> T E'
E' -> + T #1 E' | _
T -> F T'
T' -> * F #2 T' | _
F -> int #3 | (E) #4
```

How does this still work? Let's look at action `#1` (which we gave code for previously) as an example. From the code, it is clear that the action expects an `ASTNode`, a plus token, and another `ASTNode` on the top of the semantic stack. However, the production for `E'` guarantees only that the first `ASTNode` and the plus token will be on the stack. To ensure that the other `ASTNode` will be there, we must look at where `E'` appears in other productions. For the production `E -> T E'`, the parsing of the preceding `T` will provide the requisite `ASTNode`. For the production `E' -> + T #1 E'`, the execution of the `#1` action will push the `ASTNode` on the stack; so, we are fine. You will be employing the same trick in PA3, except that you will be fixing the productions first, and then adding actions.

When parsing with a recursive-descent parser instead of LL(1), we need to do actions slightly differently. Recursive-descent parsers use backtracking, and when actions are added, we will need to backtrack the semantic stack along with the position in the input. We do not want actions to pop elements from the semantic stack, since undoing these pops during backtracking could be quite expensive. For example, consider action `#1` as written. Bad things will happen if we execute the action and then cannot parse the rest of the input to `E'`, since we would then need to somehow push back the elements corresponding to `+` and `T`. Instead of dealing with this hassle, we restrict actions to only peek at elements in the semantic stack and not pop any of them.

How do semantic stack elements ever get popped? At the end of a successful parse of a production (when we know we won't need to backtrack its intermediate transformations), we pop all elements pushed during the parse to that production, and then re-push the final element pushed during the parse. So, for action `#1`, we would first rewrite it to only peek at the stack instead of popping elements. Then, after successfully parsing to the following `E'` non-terminal, we would pop the semantic stack elements for `+`, `T`, `#1`, and `E'`, and then push back the element for `E'` (this assumes that each action pushes *exactly one* object on the semantic stack). Assuming appropriate behavior for ϵ (it just duplicates whatever is at the top of the stack), you can reason out this behavior and show that it does the right thing.

Note that we cannot rely on actions to perform this final popping and pushing, since they cannot perform pops. This code must be included in the parser, along with code to backtrack the semantic stack appropriately.