Summary of Syntax Directed Translation

1. Semantic records and the semantic stack

The most important concept with syntax directed translation is the assignment of semantic values (or attributes) to grammar symbols during parsing. At every parsing step, the parser handles the grammar symbols in a specific production rule. With a schematic way to bind useful semantic attributes to each symbol, the compiler can collect semantic information needed for the rest of the compilation tasks, such as type checking, memory allocation, and code generation) in a consistent and reliable manner.

The semantic attributes are collected and propagated starting with the lexical analyzer. When the lexical analyzer recognizes a number, it can immediately perform a *semantic action* to evaluate the value of the number and pass that value, in addition to the token kind (i.e. number) , to the parser. Similarly, when the lexical analyzer recognizes an identifier, it can immediately perform a semantic action to either pass the lexeme of the identifier to the parser or to enter the lexeme to a name table (shared between the lexical analyzer and the parser) and pass the index to the parser.

The implementation of semantic-attributes propagation is through *semantic records* (associated with the grammar symbols). The Lex tool uses the variable yylval to associate the semantic record (holding the semantic value such as the value of a number) with the recognized token. This variable yylval will be overwritten when the next token is recognized. With bottom parsing based on the YACC/Bison tool, every time the parser (yyparse()) performs a shift (on a token), the value of yylval is pushed to *the semantic stack*.

The parsing stack (for bottom-up parsing) has a one-to-one correspondence with the semantic stack. Both stacks grow and shrink in synch. At the time of a reduce action, the semantic actions appended to the corresponding production rule in the YACC/bison program are performed. The semantic actions can make references to the semantic records associated with the grammar symbols in the production rule through the parameters ($1, $2, etc). The semantic record associated with the left-hand side nonterminal is referenced as $$.

We may want to pass different kind of attributes, such as numerical values, pointers, and so on, in different semantic actions. Hence, it is common to re-declare the type of semantic records as structures instead of integers. (See YACC/Bison manuals and tutorials to find out how to do this.)

2. Examples of syntax-directed translation

In class, we illustrated two examples. The first is a “calculator” that evaluates an arithmetic expression with integers as operands. Through this example, we illustrated the importance of using the correct recursive production rules, e.g. left-recursive instead of right-recursive for + and – expressions, such that

the correct associativity is implemented when we evaluate the expression through semantic actions. This is due to the way syntax-directed translation is implemented as discussed above.

We also illustrated the need to introduce new non-terminals in order to impose the precedence rules that separate + and – from * and / operators. The YACC/Bison tool provide directives such as %left to specify associativity and precedence rules, but it is preferable to use non-terminals to avoid confusion when we need to examine the state diagram.

The second example is a simplified Javascript that consists of a list of assignment statements. When we do not consider branch statements such as IF and loops, we can interpret the statements one by one as we parse them, without waiting to finish parsing the whole program. A correct program would initialize the value of a variable before using it as an operand in an expression. We can use a symbol table to store the values of variables, updating the values as we interpret the statements one by one.

Each entry in the symbol table would have two attributes, one being the identifier and the other its value. The identifier can be either represented by its lexeme or by its index in yet another table called the name table. If we represent the identifiers by their lexemes, then the lexical analyzer must pass the lexeme of each recognized identifier token to the parser. A more convenient way might be to share a name table between the lexical analyzer and the parser. The lexical analyzer would look up the name table to determine the index for each identifier and pass the index to the parser. In the parser, when id = <expr> is reduced to <stmt>, the semantic action stores the value of <expr> to the symbol table entry for the id.

In the more general case, we may have branch statements such as IF and loops. We are no longer able to interpret the statements on the fly. (You can think what difficulties there will be.) Instead, we will need to build an internal representation (IR), e.g. an AST, on the fly, using the semantic actions associated with production rules. When the parsing completes, we traverse the AST to execute the statements. This general scheme would work as follows, although we don't discuss IF statements and loops in details.

We build the AST incrementally through the semantic actions, as shown in class, by calling two utility routines mkleaf(leaf_kind, value) and mknode(operator, left_operand, right_operand). These two routines create leaves and internal nodes (for the AST) dynamically. Each returns a reference (i.e. a pointer) to the created node (a leaf or an internal node). We let the semantic record contain a field that is a pointer to a node. We let $$.ptr point to the node we just created. When we reduce E → E + T, we let $$.ptr = mknode(+, $1.ptr, $3.ptr) , for example.

For IF statements and loops, we need to design special nodes to store all the information we need. For example, for IF statements, we would need a field for the Boolean condition, another field for the pointer to the list of statements in the true branch, and yet one more field for the false branch.

The value in mkleaf(leaf_kind, value) depends on our design. For our simple language, if the leaf is an identifier, we can let value be the index in a name table for the name of the identifier. If the leaf is a number we let value be the numerical value of the number. Every time the lexical analyzer recognizes an

identifier, it must look up the name table to determine the index for the identifier. In a real compiler, a hash table is usually the way to implement it.

After yyparse() returns, we have an AST and a name table. We can now execute a function that traverses the AST in depth-first search and interpret the given program by performing the operations indicated in the AST. A new table, called the symbol table (symtab), is used to store the current value for each identifier.

When we visit a leaf that is an identifier appearing in the left-hand side of an assignment statement, we look up the symbol table to find its location. If it is not found there, we assign a location to the identifier. We then place the computed value of the right-hand side expression into the symbol table entry for the left-hand side identifier.

When we visit a leaf that is an identifier appearing in a right-hand side expression, we look up the symbol table for retrieve the current value of the identifier.

3. Upcoming homework assignment

In the upcoming homework 3, we will enhance the parser built for homework 2 by implementing the interpretation scheme discussed above in YACC/Bison. The difference from the above is the addition of declarations and document.print statements. The existence of declaration statements allow us to impose a semantic rule that a variable must be declared before referenced. We implement document.print by calling printf to standard output. Details will be forthcoming.