Syntax Directed Translation

Handout written by Maggie Johnson and revised by Julie Zelenski.

Syntax-directed translation refers to a method of compiler implementation where the source language translation is completely driven by the parser. In other words, the parsing process and parse trees are used to direct semantic analysis and the translation of the source program. This can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called *attribute grammars*.

We augment a grammar by associating *attributes* with each grammar symbol that describes its properties. An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register—whatever information we need. For example, variables may have an attribute "type" (which records the declared type of a variable, useful later in type-checking) or an integer constant may have an attribute "value" (which we will later need to generate code).

With each production in a grammar, we give semantic rules or *actions*, which describe how to compute the attribute values associated with each grammar symbol in a production. The attribute value for a parse node may depend on information from its children nodes below or its siblings and parent node above.

Consider this production, augmented with a set of actions that use the "value" attribute for a digit node to store the appropriate numeric value. Below, we use the syntax `x.a` to refer to the attribute `a` associated with symbol `x`.

```
digit    -> 0   {digit.value = 0}
         |  1   {digit.value = 1}
         |  2   {digit.value = 2}
         ...
         |  9   {digit.value = 9}
```

Attributes may be passed up a parse tree to be used by other productions:

```
int₁     -> digit       {int₁.value = digit.value}
         |  int₂ digit  {int₁.value = int₂.value*10 + digit.value}
```

(We are using subscripts in this example to clarify which attribute we are referring to, so int_1 and int_2 are different instances of the same non-terminal symbol.)

There are two types of attributes we might encounter: synthesized or inherited. *Synthesized* attributes are those attributes that are passed up a parse tree, i.e., the left-side attribute is computed from the right-side attributes. The lexical analyzer usually supplies the attributes of terminals and the synthesized ones are built up for the nonterminals and passed up the tree.

$$X \rightarrow Y_1 Y_2 ... Y_n$$

$$X.a = f(Y_1.a, Y_2.a, ...Y_n.a)$$

int   value = 4*10 + 2 = 42

int value = 4   digit value = 2

digit value = 4   2

4
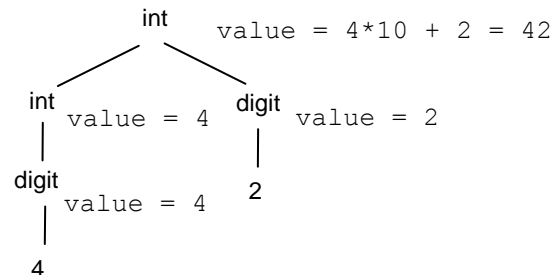
Inherited attributes are those that are passed down a parse tree, i.e., the right-side attributes are derived from the left-side attributes (or other right-side attributes). These attributes are used for passing information about the context to nodes further down the tree.

$$X \rightarrow Y_1 Y_2 ... Y_n$$

$$Y_k.a = f(X.a, Y_1.a, Y_2.a, ..., Y_{k-1}.a, Y_{k+1}.a, ..., Y_n.a)$$

Consider the following grammar that defines declarations and simple expressions in a Pascal-like syntax:

```
P  ->   DS
D  ->   var V; D | ε
S  ->   V := E; S | ε
V  ->   x | y | z
```

Now we add two attributes to this grammar, **name** and **dl**, for the name of a variable and the list of declarations. Each time a new variable is declared, a synthesized attribute for its name is attached to it. That name is added to a list of variables declared so far in the synthesized attribute dl that is created from the declaration block. The list of variables is then passed as an inherited attribute to the statements following the declarations for use in checking that variables are declared before use.

```
P    ->  DS              { S.dl = D.dl }
D₁   ->  var V; D₂       { D₁.dl = addlist(V.name, D₂.dl) }
     |    ε              { D₁.dl = NULL }
S₁   ->  V := E; S₂      { check(V.name, S₁.dl); S₂.dl = S₁.dl }
     |    ε
V    ->  x               { V.name = 'x' }
     |   y               { V.name = 'y' }
     |   z               { V.name = 'z' }
```
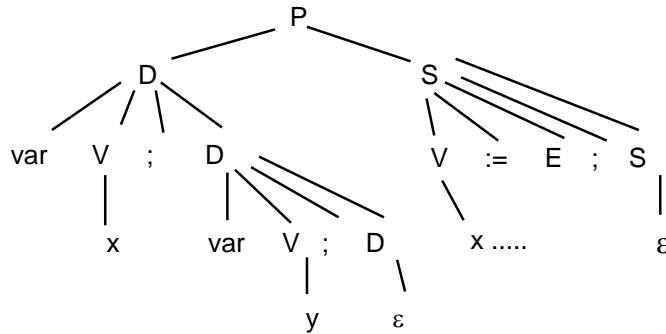
If we were to parse the following code, what would the attribute structure look like?

```
var x;
var y;

x := ...;
y := ...;
```


Typically the way we handle attributes is to associate with each symbol some sort of structure with all the necessary attributes, and we can pick out the attribute of interest by fieldname.

```
typedef struct _attribute {
   char *name;
   struct _attribute *list;
} attribute;

   P -> DS              { $2.list = $1.list }
   D -> var V; D        { $$.list = add_to_list($2.name, $4.list) }
      | ε               { $$.list = NULL }
   S -> V := E; S       { check($1.name, $$.list); $5.list = $$.list }
      | ε
   V -> x               { $$.name = 'x' }
      | y               { $$.name = 'y' }
      | z               { $$.name = 'z' }
```

Top-Down SDT

We can implement syntax-directed translation in either a top-down or a bottom-up parser and we'll briefly investigate each approach. First, let's look at adding attribute information to a hand-constructed top-down recursive-descent parser. Our example will be a very simple FTP client, where the parser accepts user commands and uses a syntax-directed translation to act upon those requests. Here's in the grammar we'll use, already in an LL(1) form:

```
Session        –>      CommandList T_QUIT
CommandList    –>      Command CommandList | ε
Command        –>      Login | Get | Logout
Login          –>      User Pass
User           –>      T_USER T_IDENT
Pass           –>      T_PASS T_IDENT
Get            –>      T_GET T_IDENT MoreFiles
MoreFiles      –>      T_IDENT MoreFiles | ε
Logout         –>      T_LOGOUT
```

Now, let's see how the attributes, such as the username, filename, and connection, can be passed around during the parsing. This recursive-descent parser is using the lookahead/token-matching utility functions from the top-down parsing handout.

```
static void ParseCommands()
{
   Connection *conn = NULL;
   int next;

   while ((next = GetLookahead()) != T_QUIT) {
      switch (next) {
         case T_USER:
            conn = ParseLogin(); break;
         case T_GET:
            ParseGet(conn); break;
         case T_LOGOUT:
            ParseLogout(conn); conn = NULL; break;
         default:
            ReportParseFailure("command expected", yytext);
      }
   }
}

static Connection *ParseLogin() // returns attribute of opened connection
{
   char *user = ParseUser();
   char *pass = ParsePassword();
   return Login(user, pass); // uses attributes passed up from below
}

static char *ParseUser() // returns attribute of username given
{
   MatchToken(T_USER);
   char *str = ParseIdentifier(); // gets name from identfier child node
   MatchToken('\n');
   return str;
```

```
   }

   static char *ParsePassword() // similar to username
   {
      MatchToken(T_PASS);
      char *str = ParseIdentifier();
      MatchToken('\n');
      return str;
   }

   static char *ParseIdentifier()
   {
      char *str = NULL;

      if (GetLookahead() == T_IDENT) {
         str = strdup(yytext);
         MatchToken(T_IDENT);
      }

      return str; // returns NULL on error
   }

   static void ParseGet(Connection *conn) // receives conn from above
   {
      MatchToken(T_GET);
      char *filename = ParseIdentifier();
      MatchToken('\n');
      Transfer(conn, filename);// reports error if conn NULL
   }

   static void ParseLogout(Connection *conn) // receives conn from above
   {
      MatchToken(T_LOGOUT);
      MatchToken('\n');
      Logout(conn); // reports error if conn NULL
   }
```

During processing of the Login command, the parser gathers the username and password returned from the children nodes and uses that information to create a new connection attribute to pass up the tree. In this situation the username, password, and connection are all acting as *synthesized* attributes, working their way from the leaves upward to the parent. That open connection is saved and then later passed downward into other commands. The connection is being used as an *inherited* attributed when processing Get and Logout, those commands are receiving information from the parent about parts of the parse that have already happened (its left siblings).

Bottom-Up SDT

Here is a simple expression grammar that has associativity and precedence already built in.

```
E'      –>      E
E       –>      T | E A T
T       –>      F | T M F
F       –>      (E) | int
A       –>      + | -
M       –>      * | /
```

During the bottom-up parse, as we push symbols on to the parse stack, we will associate with each operand/expression symbol (E, T, F, etc.) an integer value. For each operator (A, M) we will store the operator code. When performing a reduction, we will synthesize the attribute for the left-side nonterminal from the attributes of the right side symbol, the handle that is currently on top of the stack. See how the associated actions below will evaluate the arithmetic expression during parsing?

```
E'  -> E            { printf("%d\n", E.val); }

E   -> T            { E.val = T.val; }
   | E A T          { switch(A.op) {
                        case ADD: E.val = E.val + T.val; break;
                        case SUB: E.val = E.val - T.val; break; }
                    }
T   -> F            { T.val = F.val;}
   | T M F          { switch(M.op) {
                        case MUL: T.val = T.val * F.val; break;
                        case DIV: T.val = T.val / F.val; break; }
                    }
F   -> (E)          { F.val = E.val; }
   | int            { F.val = int.val; }

A   -> +            { A.op = ADD; }
   | -              { A.op = SUB; }

M   -> *            { M.op = MUL; }
   | /              { M.op = DIV; }
```

The attribute value of terminals is assumed to have been assigned by the scanner. The attribute values for nonterminals are explicitly assigned in the parser action code. The symbol attributes can usually be stored along with the symbol itself in the bottom-up parse stack, which is mighty convenient. However, given the way a bottom-up parser constructs the leaves first and works its way up to the parent, it is trivial to support synthesized attributes (like those needed in the expression parser), but more awkward to allow for inherited attributes.

Attributes and `yacc`

It doesn't take much to transform the above abstract example into legitimate yacc code. `$1`, `$2`, etc. are all used to access the attribute of the nth token on the right side of the production. The global variable `yylval` is set by the scanner and that value is saved with the token when placed on the parse stack. When a rule is reduced, a new state is placed on the stack, the default behavior is to just copy the attribute of `$1` for that new state, this can be controlled by assigning to `$$` in the action for the rule. By default the attribute type is an integer, but can be extended to allow for more storage of diverse types using the `%union` specification in the `yacc` input file. Here's a simple infix calculator that shows all these techniques in conjunction. It parses ordinary arithmetic expressions, uses a symbol table for setting and retrieving values of simple variables, and even has some primitive error-handling.

```
%{
    static int Lookup(const char *name);
    static void Store(const char *name, int val);
%}

%union {
    int intVal;
    char name[32];
}

%token <intVal>  T_Int
%token <name>    T_Identifier

%type  <intVal>  E

%left  '+'   '-'
%left  '*'   '/'
%right  U_minus

%%

S       :    Stmt   | S Stmt
        ;

Stmt    :  T_Identifier '=' E '\n'
                { Store($1, $3); printf("%d\n", $3);}
        |  E '\n'
                { printf("%d\n", $1);}
        |  error '\n'
                { printf("Discarding malformed expression.\n");}
        ;

E       :  E '+' E
            { $$ = $1 + $3;}
        |  E '-' E
            { $$ = $1 - $3;}
        |  E '*' E
            { $$ = $1 * $3;}
        |  E '/' E
            { if ($3 == 0) {
                printf("divide by zero\n");
                $$ = 0;
```

```
            } else
              $$ = $1 / $3;}
        |  '(' E ')'
           { $$ = $2;}
        |  '-' E      %prec U_minus
           { $$ = -$2;}
        |  T_Int
           { $$ = $1;}
        |  T_Identifier
           { $$ = Lookup($1);}
        ;
   %%

   /* Store/Lookup functions omitted for clarify
      They are just ordinary hash table operations */
```

Here is the scanner to go with:

```
   %{
      #include "y.tab.h"
   %}

   %%

   [0-9]+       { yylval.intVal = atoi(yytext); return T_Int;}
   [a-zA-Z]+    { strncpy(yylval.name, yytext, 32); return T_Identifier;}
   [-+*/()\n=]  { return yytext[0];}
   [ \t]*       { /* ignore whitespace */ }
```

Check out the online manual for more details about these and any other features of `yacc/bison`.

Bibliography

A. Aho, R. Sethi, J. Ullman, <u>Compilers: Principles, Techniques, and Tools</u>. Reading, MA: Addison-Wesley, 1986.

J.P. Bennett, <u>Introduction to Compiling Techniques</u>. Berkshire, England: McGraw-Hill, 1990.

A. Pyster, <u>Compiler Design and Construction</u>. New York, NY: Van Nostrand Reinhold, 1988.