Syntax-directed translation

Handout written by Maggie Johnson and revised by me.

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 constants 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:

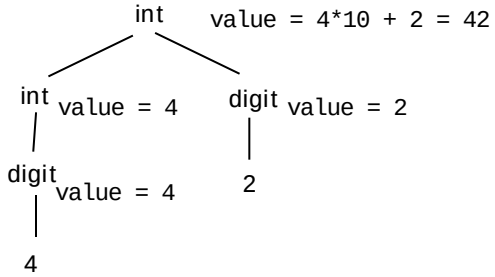
```
int1     -> digit         {int1.value = digit.value}
          | int2 digit    {int1.value = int2.value*10 + digit.value}
```

(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. Values for the attributes of terminals are usually supplied by the lexical analyzer and the synthesized ones are passed up from there.

$X \rightarrow Y_1 Y_2 ... Y_n$

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



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 | ∂
S  –>  V := E; S | ∂
V  –>  x | y | z
```

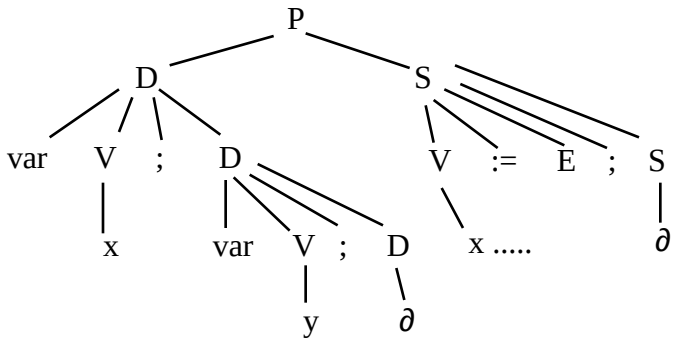
Now we add two attributes to this grammar, name and dl. Each time a new variable is declared, a synthesized attribute for its name is attached to it. That name is added to a list of variables declared so far in the synthesized attribute dl that is created from the declaration block. The list of variables 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)} | {D1.dl = NULL}
S1  –>  V := E; S2 | ∂  {check(V.name, S1.dl); S2.dl = S1.dl}
V   –>  x | y | z      {V.name = 'x'} | {V.name = 'y'} | {V.name = 'z'}
```

If we were to parse the following code, what would the attribute structure look like?

```
var x;
var y;

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



Here's a little more concrete syntax for how we might do such a thing in a parser generator:

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

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

Top-down SDT

We can implement syntax-directed translation in either a top-down or a bottom-up parser. Let's see an example of each approach. First, an LL(1) expression grammar for the usual binary arithmetic expressions with appropriate precedence already in place:

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

Now, let's attach actions to each production to calculate the value of the expression being parsed. This approach uses a stack to keep track of the intermediate results when parsing:

```
E -> TE'

E' -> A T E'     { rhs = PopOperand();
                   lhs = PopOperand();
                   switch (PopOperator()) {
                     case ADD: PushOperand(lhs+rhs); break;
                     case SUB: PushOperand(lhs-rhs); break;
                   }
                 }
   |             { /* empty, do nothing */ }

T -> FT'

T' -> M F T'     { rhs = PopOperand();
                   lhs = PopOperand();
                   switch (PopOperator()) {
                     case MUL: PushOperand(lhs*rhs); break;
                     case DIV: PushOperand(lhs/rhs); break;
                   }
                 }
   |             { /* empty, do nothing */ }

A -> +           { PushOperator(ADD);}
   | -           { PushOperator(SUB);}

M -> *           { PushOperator(MUL);}
   | /           { PushOperator(DIV);}

F -> (int        { PushOperand($1);}
   | (E)         { /* handled during parsing of E */ }
```

As we complete a parse of the right-hand side, we take the action indicated for that production. The `$` operator is used to set and reference attributes in action code. `$1` refers to the attribute variable associated with the first symbol on the right side; `$2` the second symbol, etc.

Trace what happens when we parse `2 + 4 * 3`:

Parser action	Parse stack	Operand stack	Operator stack
Predict E –> TE'	E$		
Predict T –> FT'	TE'$		
Predict F –> int	FT'E'$		
Match int	intT'E'$		
Predict T' –> ∂	T'E'$	2	
Predict E' –> ATE'	E'$	2	
Predict A –> +	ATE'$	2	
Match +	+TE'$	2	
Predict T –> FT'	TE'$	2	+
Predict F –> int	FT'E'$	2	+
Match int	intT'E'$	2	+
Predict T' –> MFT'	T'E'$	4 2	+
Predict M –> *	MFT'E'$	4 2	+
Match *	*FT'E'$	4 2	+
Predict F –> int	FT'E'$	4 2	* +
Match int	intT'E'$	4 2	* +
Predict T' –> ∂	T'E'$	3 4 2	* +
Predict E' –> ∂	E'$	12 2	+
Success!	$	14	

Bottom-up SDT

Here is an expression grammar with attributes and actions built for a bottom-up parser. Again, the expression is evaluated as it is parsed.

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

In this attributed grammar we use the actual attribute to store the value as the sentence is parsed.

```
E'  -> E        { printf("%d\n", $1); }

E   -> T        { $$ = $1; }
     | E A T    { switch($2) {
                     case ADD: $$ = $1 + $3; break;
                     case SUB: $$ = $1 - $3; break;
                  }
T   -> F        { $$ = $1;}
     | T M F    { switch($2) {
                     case MUL: $$ = $1 * $3; break;
                     case DIV: $$ = $1 / $3; break;
```

```
                }
F   -> (E)      { $$ = $2; }
     | int      { $$ = $1; }

A   -> +        { $$ = ADD; }
     | -        { $$ = SUB; }

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

As above, the `$` operator is used to set and reference attributes in action code. `$$` means the attribute associated with the non-terminal on the left side. The attribute value of terminals is assigned by the scanner and can be referenced in the action code. The attribute values of non-terminals must be explicitly assigned in action code.

Exercise for the reader: Trace what happens when parsing 2 * 4 + 3 and 2 + 4 * 3 in the bottom-up parser.

Attributes and Yacc

Access to attributes in yacc looks pretty much as shown above in the abstract example, but let's make it more concrete with a real yacc specification. The syntax $1, $2 is 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 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

%nonassoc '='
%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("Error, discarding\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 hashtable 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 */ }
```

We'll try to talk through the interesting features of this parser in lecture. Check out the online manual for more details about these and any other features of yacc and bison.

Bibliography

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

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

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