Syntax-Directed Translation

ASU Textbook Chapter 5.1–5.6, 4.9

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

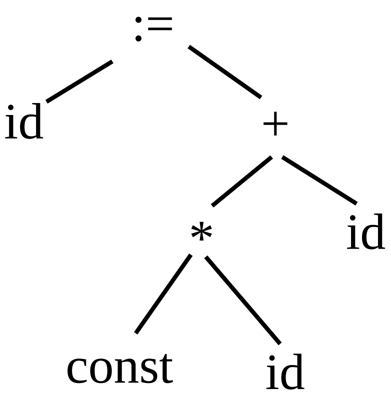
`http://www.iis.sinica.edu.tw/~tshsu`

What is syntax-directed translation?

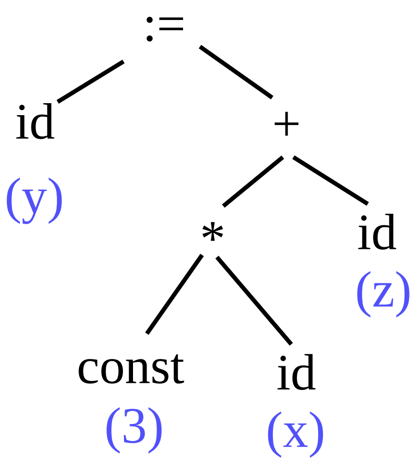
- **Definition:**
  - **The compilation process is driven by the syntax.**
  - **The semantic routines perform interpretation based on the syntax structure.**
  - **Attaching attributes to the grammar symbols.**
  - **Values for attributes are computed by semantic rules associated with the grammar productions.**

Example: Syntax-directed translation

- **Example in a parse tree:**
  - **Annotate the parse tree by attaching semantic attributes to the nodes of the parse tree.**
  - **Generate code by visiting nodes in the parse tree in a given order.**
  - **Input:** $y := 3 * x + z$



parse tree



annotated parse tree

Syntax-directed definitions

- **Each grammar symbol is associated with a set of attributes.**
    - **Synthesized attribute : values computed from its children or associated with the meaning of the tokens.**
    - **Inherited attribute : values computed from parent and/or siblings.**
    - **general attribute: values can be depended on the attributes of any nodes.**

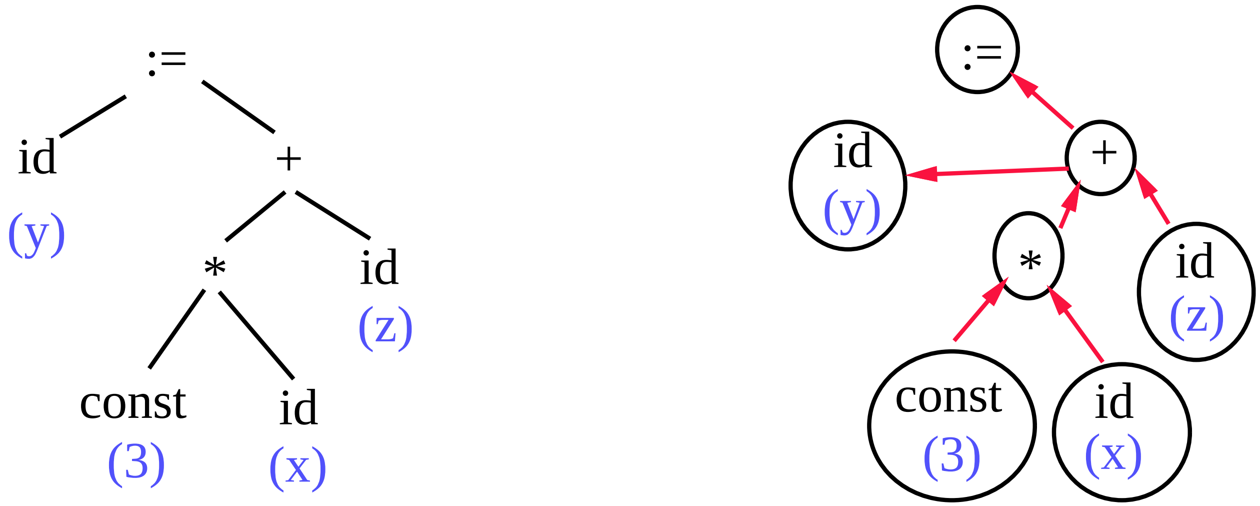
Format for writing syntax-directed definitions

Production	Semantic rules
$L \to E$	$\mathrm{print}(E.val)$
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to T$	$E.val := T.val$
$T \to T_1 * F$	$T.val := T_1.val * F.val$
$T \to F$	$T.val := F.val$
$F \to (E)$	$F.val := E.val$
$F \to digit$	$F.val := digit.lexval$

- **$E.val$ is one of the attributes of E.**
- **To avoid confusion, recursively defined nonterminals are numbered on the LHS.**

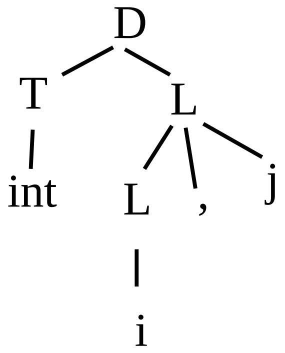
Order of evaluation (1/2)

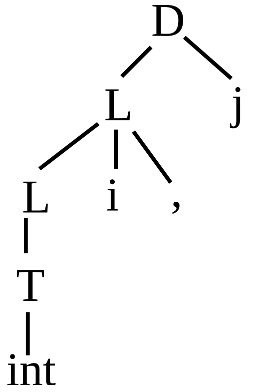
- **Order of evaluating attributes is important.**
- **General rule for ordering:**
  - **Dependency graph** :
    - *If attribute b needs attributes a and c, then a and c must be evaluated before b.*
    - *Represented as a directed graph without cycles.*
    - *Topologically order nodes in the dependency graph as n_1, n_2, . . ., n_k such that there is no path from n_i to n_j with $i > j$.*



Order of evaluation (2/2)

- **It is always possible to rewrite syntax-directed definitions using only synthesized attributes, but the one with inherited attributes is easier to understand.**
  - **Use inherited attributes to keep track of the type of a list of variable declarations.**
    - $int\ i, j$
  - **Reconstruct the tree:**
    - $D \rightarrow T L$
    - $T \rightarrow int \mid char$
    - $L \rightarrow L, id \mid id$
    - $D \rightarrow L\ id$
    - $L \rightarrow L\ id, \mid T$
    - $T \rightarrow int \mid char$



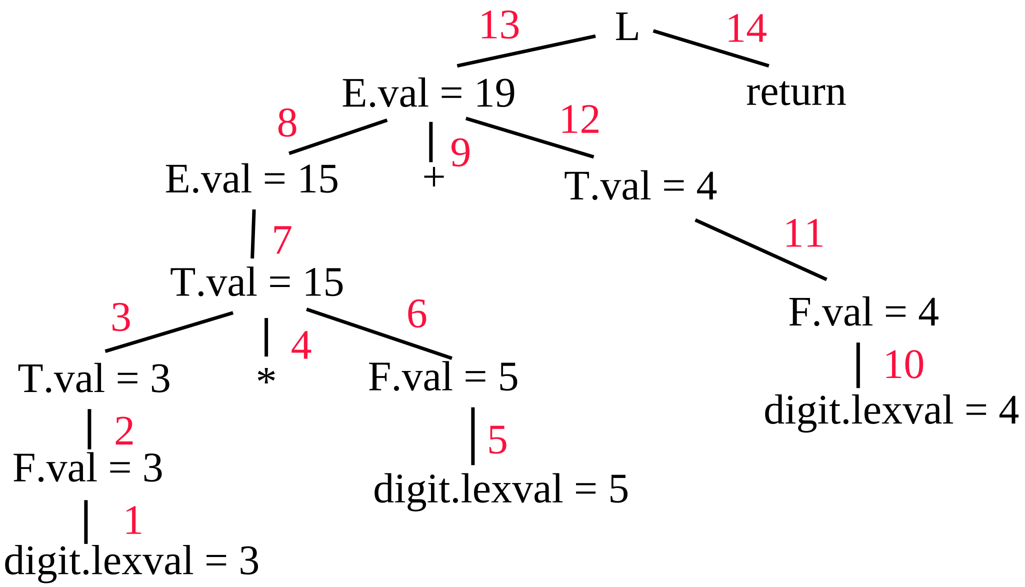


Attribute grammars

- **Attribute grammar: a grammar with syntax-directed definitions such that functions used cannot have side effects.**
  - **Side effect: change values of others not related to the return values of functions themselves.**
- **Tradeoffs:**
  - **Synthesized attributes are easy to compute, but are sometimes difficult to be used to express semantics.**
  - **Inherited and general attributes are difficult to compute, but are sometimes easy to express the semantics.**
  - **The dependence graph for computing some inherited and general attributes may contain cycles and thus not-computable.**
  - **A restricted form of inherited attributes is invented.**
    - ▷ ***L-attributes.***

S-attributed definition

- **Definition: a syntax-directed definition that uses synthesized attributed only.**
  - **A parse tree can be represented using a directed graph.**
  - **A post-order traverse of the parse tree can properly evaluate grammars with S-attributed definitions.**
  - **Bottom-up evaluation.**
- **Example of an S-attributed definition: $3 * 5 + 4$ return**



L-attributed definition

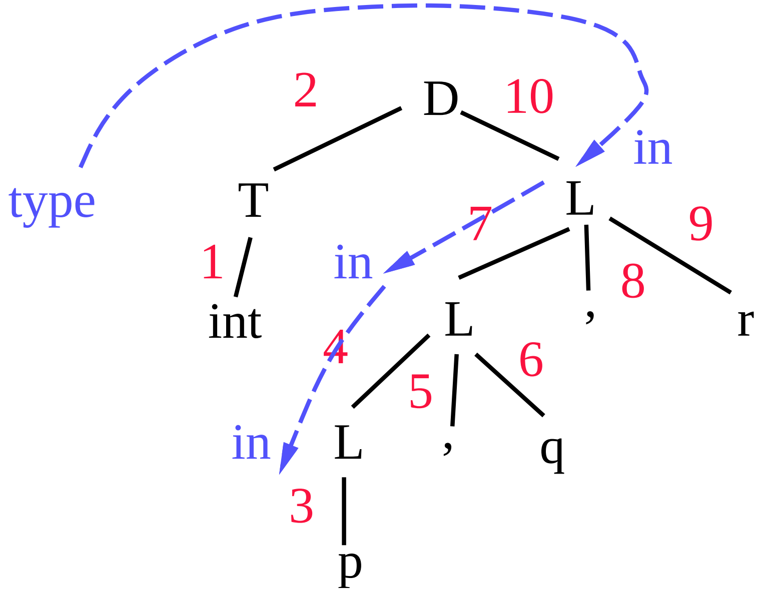
- **Definition:**
  - **Each attribute in each semantic rule for the production $A \rightarrow X_1, \cdots, X_n$ is either a synthesized attribute or an inherited attribute X_j depends only on the inherited attribute of A and/or the attributes of $X_1, \ldots, X_{j-1}$.**
  - **Every S-attributed definition is an L-attributed definition.**
- **For grammars with L-attributed definitions, special evaluation algorithms must be designed.**
- **Bottom-up evaluation of L-attributed grammars.**
  - **Can handle all $LL(1)$ grammars and most $LR(1)$ grammars.**
  - **All translation actions are taken at the right end of the production.**
- **Key observation:**
  - **L-attributes are always computable.**
    - ▷ ***Same argument as the one used in discussing Algorithm 4.1.***
  - **when a bottom-up parser reduces by the production $A \rightarrow XY$, by removing X and Y from the top of the stack and replacing them by A,**
  - **$X.s$ (the synthesized attribute of X) is on the top of the stack and thus can be used to compute $Y.in$ (the inherited attribute of Y).**

Example for L-attributed definitions

- $D \rightarrow T \ \{L.in := T.type\} \ L$
- $T \rightarrow int \ \{T.type := integer\}$
- $T \rightarrow real \ \{T.type := real\}$
- $L \rightarrow \ \{L_1.in := L.in\} \ L_1, id \ \{addtype(id.entry, L.in)\}$
- $L \rightarrow id \ \{addtype(id.entry, L.in)\}$

Parsing and dependency graph:

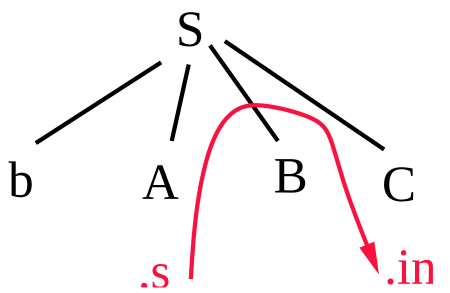
input	stack	production used
int p, q, r		
p, q, r	**int**	
p, q, r	T	$T \rightarrow int$
$, q, r$	$T \ p$	
$, q, r$	$T \ L$	$L \rightarrow id$
q, r	$T \ L \ ,$	
$, r$	$T \ L \ , \ q$	
$, r$	$T \ L$	$L \rightarrow L, id$
r	$T \ L \ ,$	
	$T \ L \ , \ r$	
	$T \ L$	$L \rightarrow L, id$
	D	$D \rightarrow TL$



Using of markers

- **Information contained in the stack can be used by replacing special markers to mark the production we are currently in.**
  - **Example 1:**

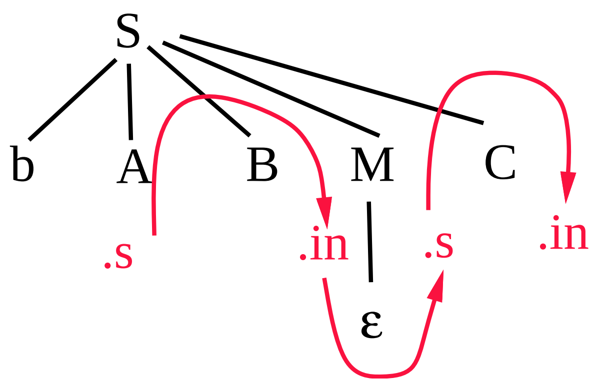
production	semantic rules
$S \rightarrow aAC$	$C.in := A.s$
$S \rightarrow bABC$	$C.in := A.s$
$C \rightarrow c$	$C.s := \cdots$
$\cdots$	$\cdots$



Same rule for the first two productions. It is difficult to tell which one and to find the position of A in the stack in each case.

  - **Example 2:**

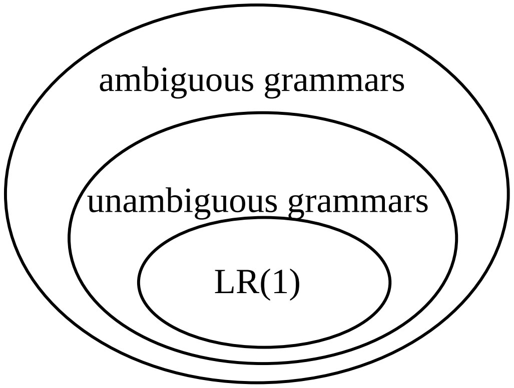
production	semantic rules
$S \rightarrow aAC$	$C.in := A.s$
$S \rightarrow bABMC$	$M.in := A.s;$ $C.in := M.s$
$C \rightarrow c$	$C.s := \cdots$
$M \rightarrow \epsilon$	$M.s := M.in$
$\cdots$	$\cdots$



A is always one place below in the stack.

- **Markers can also be used to perform error checking and other intermediate semantic actions.**

Using ambiguous grammars



- **Ambiguous grammars provides a shorter, more natural specification than any equivalent unambiguous grammars.**
- **Sometimes need ambiguous grammars to specify important language constructs.**
- **For example: declare a variable before its usage.**

```
var xyz : integer
begin
    ...
    xyz := 3;
    ...
```

Ambiguity from precedence and associativity

- **Use precedence and associativity to resolve conflicts.**
- **Example:**
  - G_1:
    - ▷ $E \longrightarrow E + E \mid E * E \mid (E) \mid id$
    - ▷ *ambiguous, but easy to understand!*
  - G_2:
    - ▷ $E \longrightarrow E + T \mid T$
    - ▷ $E \longrightarrow T * F \mid F$
    - ▷ $F \longrightarrow (E) \mid id$
    - ▷ *unambiguous, but it is difficult to change the precedence;*
    - ▷ *parse tree is much larger for G_2, and thus takes more time to parse.*
- **When parsing the following input for G_1: $id + id * id$.**
  - **Assume the input parsed so far is $id + id$.**
  - **We now see "*".**
  - **We can either shift or perform "reduce by $E \rightarrow E + E$".**
  - **When there is a conflict, say in $SLR(1)$ parsing, we use precedence and associativity information to resolve conflicts.**

Dangling-else ambiguity

- **Grammar:**
  - $S \rightarrow$<**statement**>
    $\mid if$ <**condition**> $then$ <**statement**>
    $\mid if$ <**condition**> $then$ <**statement**> $else$ <**statement**>
- **When seeing**

  $\underline{\textbf{if c then S}}$ else S

  - **there is a shift or reduce conflict;**
  - **we always favor a shift.**
  - **Intuition: favor a longer match.**

Special cases

- **Ambiguity from special-case productions:**
  - **Sometime a very rare happened special case causes ambiguity.**
  - **It is too costly to revise the grammar. We can resolve the conflicts by using special rules.**
  - **Example:**
    - $E \rightarrow E\ sub\ E\ sup\ E$
    - $E \rightarrow E\ sub\ E$
    - $E \rightarrow E\ sup\ E$
    - $E \rightarrow \{E\} \mid character$
  - **Meanings:**
    - $W\ sub\ U$: W_U.
    - $W\ sup\ U$: W^U.
    - $W\ sub\ U\ sup\ V$ is W_U^V, not ${W_U}^V$
  - **Resolve by semantic and special rules.**
  - **Pick the right one when there is a reduce/reduce conflict.**
    - *Reduce the production listed earlier.*
  - **Similar to the dangling-else case!**

YACC implementation

- **YACC can be used to implement L-attributed definitions.**
  - **Use of global variables to record the inherited values from its older siblings.**
  - **Use of STACKS to pass synthesized attributes.**
  - **It is difficult to use information passing from its parent node.**
    - ▷ *It may be possible to use the state information to pass some information.*
- **Passing of synthesized attributes is best.**
  - **Without using global variables.**
- **Cannot use information from its younger siblings because of the limitation of LR parsing.**
  - **During parsing, the STACK contains information about the older siblings.**

YACC (1/2)

- **Yet Another Compiler Compiler:**
  - **A UNIX utility for generating $LALR(1)$ parsing tables.**
  - **Convert your YACC code into C programs.**
  - **file.y** $\longrightarrow$ $\boxed{\textbf{yacc file.y}}$ $\longrightarrow$ **y.tab.c**
  - **y.tab.c** $\longrightarrow$ $\boxed{\textbf{cc y.tab.c -ly -ll}}$ $\longrightarrow$ **a.out**
- **Format:**
  - **declarations**
    - ▷ *%{ · · · %} to enclose C declarations.*
  - **%%**
  - **translation rules**
    - ▷ *<left side>: <production>*
    - ▷ *{ semantic rules }*
  - **%%**
  - **supporting C-routines.**

YACC (2/2)

- **Assume the Lexical analyzer routine is $yylex()$.**
- **When there are ambiguities:**
  - **reduce/reduce conflict: favor the one listed first.**
  - **shift/reduce conflict: favor shift, i.e., longer match!**
- **Error handling:**
  - **Example:**

```
lines: error '\n' {...}
```

    - ▷ ***When there is an error, skip until newline is seen.***
    - ▷ ***One of the reasons to use statement terminators, instead of statement separators, in language designs.***
  - $error$**: special non-terminal.**
    - ▷ ***A production with*** $error$ ***is "inserted" or "processed" only when it is in the reject state.***
    - ▷ ***It matches any sequence on the stack as if the handle "***$error \to \cdots$***" is seen.***
  - $yyerrok$**: a macro to reset error flags and make** $error$ **invisible again.**
  - $yyerror(string)$**: pre-defined routine for printing error messages.**

YACC code example (1/2)

```
%{
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#define YYSTYPE int /* integer type for YACC stack */

%}

%token NUMBER ERROR
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
```

YACC code example (2/2)

```
lines    :  lines expr '\n'        {printf("%d\n", $2);}
         |  lines '\n'
         |  /* empty, i.e., epsilon */
         |  lines error '\n' {yyerror("Please reenter:");yyerrok;}
         ;
expr     :  expr '+' expr    { $$ = $1 + $3; }
         |  expr '-' expr    { $$ = $1 - $3; }
         |  expr '*' expr    { $$ = $1 * $3; }
         |  expr '/' expr    { $$ = $1 / $3; }
         |  '(' expr ')'     { $$ = $2; }
         |  '-' expr  %prec UMINUS  { $$ = - $2; }
         |  NUMBER       { $$ = atoi(yytext);}
         ;

%%
#include "lex.yy.c"
```

Included LEX program

```
%{
%}
Digit          [0-9]
IntLit         {Digit}+
%%
[ \t] {/* skip white spaces */}
[\n] {return('\n');}
{IntLit}                             {return(NUMBER);}
"+"                                  {return('+');}
"-"                                  {return('-');}
"*"                                  {return('*');}
"/"                                  {return('/');}
.         {printf("error token <%s>\n",yytext); return(ERROR);}
%%
```

YACC rules

- **Can assign associativity and precedence.**
  - **in increasing precedence**
  - **left/right or non-associativity**
    - *Dot products of vectors has no associativity.*
- **Semantic rules: every item in the production is associated with a value.**
  - **YYSTYPE: the type for return values.**
  - **\$\$: the return value if the production is reduced.**
  - **\$$i$: the return value of the ith item in the production.**

In-production actions

- **Actions can be inserted in the middle of a production, each such action is treated as a nonterminal.**
  - **Example:**

    ```
    expr    :  expr { perform some semantic actions} ’+’ expr
               {$$ = $1 + $4; }
    ```

    is equivalent to

    ```
    expr   :  expr $ACT ’+’ expr {$$ = $1 + $4;}
    $ACT   : { perform some semantic actions}
    ```

- **Avoid in-production actions.**
  - **Replace them by markers.**
    - ▷ ***ϵ-productions can easily generate conflicts.***
  - **Split the production.**

    ```
    expr    :  exprhead exptail {$$ = $1 + $2;}
    exphead :  expr { perform some semantic actions; $$ = $1;}
    exptail : ’+’  expr {$$ = $2;}
    ```

    - ▷ ***May generate some conflicts.***
    - ▷ ***May be difficult to specify precedence and associativity.***

YACC programming styles

- **Keep the right hand side of a production short.**
  - Better to have less than 4 symbols.
- **Language issues.**
  - Watch out C-language rules.
    - ▷ *goto*
  - Some C-language reserved words are used by YACC.
    - ▷ *union*
  - Some YACC pre-defined routines are macros, not procedures.
    - ▷ *yyerrok*
- **Try to find some unique symbols for each production.**
  - array → ID [elist]
  -
    - ▷ $array \rightarrow aelist\ ]$
    - ▷ $aelist \rightarrow aelist, ID \mid ahead$
    - ▷ $ahead \rightarrow ID\ [\ ID$

Limitations of syntax-directed translation

- **Limitation of syntax-directed definitions: Without using global data to create side effects, some of the semantic actions cannot be performed.**
- **Example:**
    - **Checking whether a variable is defined before its usage.**
    - **Checking the type and storage address of a variable.**
    - **Checking whether a variable is used or not.**
    - **Need to use a symbol table: global data to show side effects of semantic actions.**
- **Common approach in using global variables:**
    - **A program with too many global variables is difficult to understand and maintain.**
    - **Restrict the usage of global variables to essential ones and use them as objects.**
        - ▷ ***Symbol table.***
        - ▷ ***Labels for GOTO's.***
        - ▷ ***Forwarded declarations.***
    - **Use syntax-directed definitions as much as you can.**