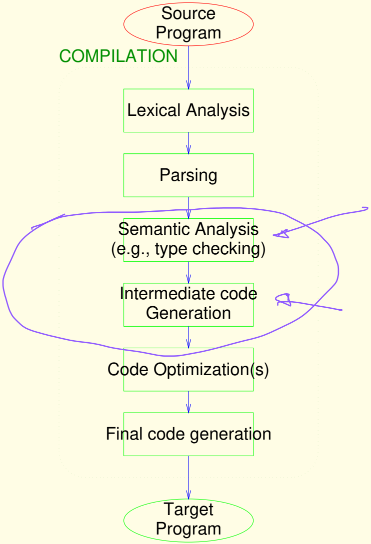
Compilation



Syntax-Directed Translation

Technique used to build semantic information for large structures, based on its syntax.

In a compiler, *Syntax-Directed Translation* is used for

- Constructing Abstract Syntax Tree
- Type checking
- Intermediate code generation

The Essence of Syntax-Directed Translation

The semantics (*meaning*) of the various constructs in the language is viewed as *attributes* of the corresponding grammar symbols.

Example: Sequence of characters `495`

- grammar symbol `TOK_INT`
- meaning ≡ integer *495* ← value
- is an attribute of `TOK_INT`(`yylval.int_val`).

Attributes are associated with ***Terminal*** as well as ***Nonterminal*** symbols.

An Example of Syntax-Directed Translation

$$\boxed{\begin{array}{lcl} E & \longrightarrow & E * E \\ E & \longrightarrow & E + E \\ E & \longrightarrow & \cancel{\mathbf{id}}\ \text{int} \end{array}}$$

$$\begin{array}{lcll} E & \longrightarrow & E_1 * E_2 & \{E.val := E_1.val * E_2.val\} \\ E & \longrightarrow & E_1 + E_2 & \{E.val := E_1.val + E_2.val\} \\ E & \longrightarrow & \mathsf{int} & \{E.val := \mathsf{int}.val\} \end{array}$$

Syntax-Directed Definitions with yacc

1 2 3

E	$\longrightarrow$	E_1 * E_2	$\{E.val := E_1.val * E_2.val\}$
E	$\longrightarrow$	E_1 + E_2	$\{E.val := E_1.val + E_2.val\}$
E	$\longrightarrow$	int	$\{E.val :=$ int$.val\}$

$\$i$

E :	E MULT E	$\{\$\$.val = \$1.val * \$3.val\}$
E :	E PLUS E	$\{\$\$.val = \$1.val + \$3.val\}$
E :	INT	$\{\$\$.val = \$1.val\}$

Another Example of Syntax-Directed Translation

$$\begin{array}{lcl} Decl & \longrightarrow & Type\ VarList \\ Type & \longrightarrow & \ldots \\ VarList & \longrightarrow & \mathsf{id}\ ,\ VarList \\ VarList & \longrightarrow & \mathsf{id} \end{array}$$

"int" | "float"

TYPE_INT

$$\begin{array}{lcll} Decl & \longrightarrow & Type\ VarList & \{VarList.type := Type.type\} \\ Type & \longrightarrow & \ldots & \{Type.type := \ldots\} \\ VarList & \longrightarrow & \mathsf{id}\ ,\ VarList_1 & \{VarList_1.type := VarList.type; \\ & & & \ \ \mathsf{id}.type := VarList.type\} \\ VarList & \longrightarrow & \mathsf{id} & \{\mathsf{id}.type := VarList.type\} \end{array}$$

"int"

TYPE_INT;

Attributes

- *Synthesized* Attribute: Value of the attribute computed from the values of attributes of grammar symbols on RHS.
  - Example: *val* in Expression grammar
- *Inherited* Attribute: Value of attribute computed from values of attributes of the LHS grammar symbol.
  - Example: *type* of *VarList* in declaration grammar

Syntax-Directed Definition

Actions associated with each production in a grammar.

For a production $A \longrightarrow X\ Y$, actions may be of the form:

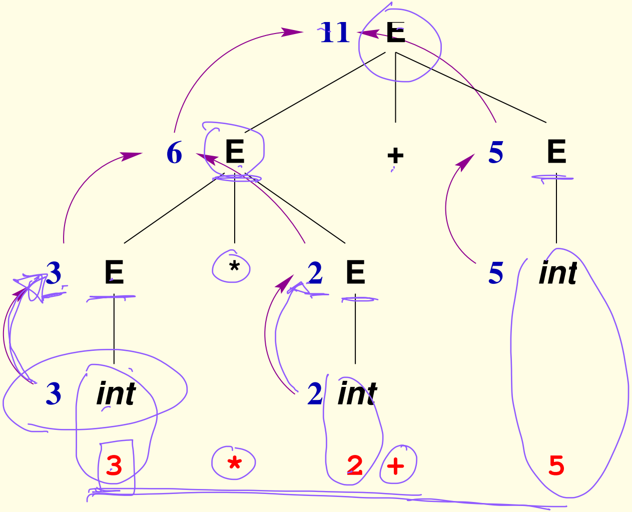
- $A.attr := f(X.attr', Y.attr'')$ for synthesized attributes
- $Y.attr := f(A.attr', X.attr'')$ for inherited attributes

Synthesized Attributes: An Example

$$\boxed{\begin{array}{lcl} E & \longrightarrow & E * E \\ E & \longrightarrow & E + E \\ E & \longrightarrow & \mathrm{int} \end{array}}$$

$$\begin{array}{lcll} E & \longrightarrow & E_1 * E_2 & \{E.val := E_1.val * E_2.val\} \\ E & \longrightarrow & E_1 + E_2 & \{E.val := E_1.val + E_2.val\} \\ E & \longrightarrow & \mathrm{int} & \{E.val := \mathrm{int}.val\} \end{array}$$

Information Flow for Synthesized Attributes

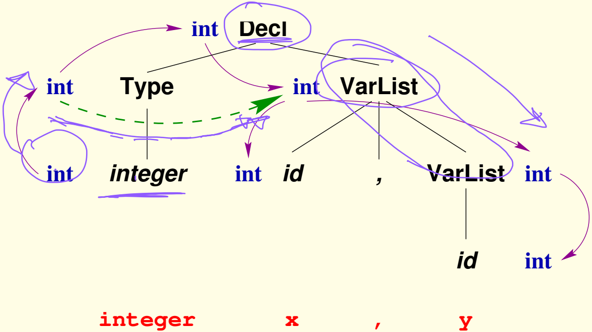


Another Example of Syntax-Directed Translation

$$\begin{array}{rcl} Decl & \longrightarrow & Type\ VarList \\ Type & \longrightarrow & \mathsf{integer} \\ Type & \longrightarrow & \mathsf{float} \\ VarList & \longrightarrow & \mathsf{id}\ ,\ VarList \\ VarList & \longrightarrow & \mathsf{id} \end{array}$$

$$\begin{array}{rcll} Decl & \longrightarrow & Type\ VarList & \{VarList.type := Type.type\} \\ Type & \longrightarrow & \mathsf{integer} & \{Type.type := int\} \\ Type & \longrightarrow & \mathsf{float} & \{Type.type := float\} \\ VarList & \longrightarrow & \mathsf{id}\ ,\ VarList_1 & \{VarList_1.type := VarList.type; \\ & & & \quad \mathsf{id}.type := VarList.type\} \\ VarList & \longrightarrow & \mathsf{id} & \{\mathsf{id}.type := VarList.type\} \end{array}$$

Information Flow for Inherited Attributes

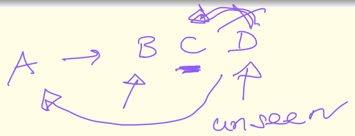


Var List →

VarList id

Varlist → id

Attributes and Definitions



- **S-Attributed Definitions:** Where all attributes are *synthesized*.
- **L-Attributed Definitions:** Where all *inherited* attributes are such that their values depend only on
  - inherited attributes of the parent, and
  - attributes of left siblings

Attributes and Top-down Parsing

- *Inherited:* analogous to function arguments
- *Synthesized:* analogous to return values

L-attributed definitions mean that argument to a parsing function is

- argument of the calling function, or
- return value/argument of a previously called function

$A \to B\ C\ D$

```
parse_A(int a){
    b= parse_B(a);
    c= parse_C(a,b);
    d= parse_D(b,c);
    return b+d;
}
parse_B(int x){
    return x+2;
}
```

Synthesized Attributes and Bottom-up Parsing

Keep track of attributes of symbols while parsing.

- Keep a stack of attributes corresponding to stack of symbols.
- Compute attributes of LHS symbol while performing reduction (*i.e.*, while pushing the symbol on symbol stack)

Synthesized Attributes and Bottom-Up Parsing

E	$\longrightarrow$	$E+E$
E	$\longrightarrow$	$E*E$
E	$\longrightarrow$	int

Parsing

STACK	INPUT STREAM	ATTRIBUTES
\$	3 * 2 + 5 \$	\$
\$ int	* 2 + 5 \$	\$ 3
\$ E	* 2 + 5 \$	\$ 3
\$ E *	2 + 5 \$	\$ 3 $\perp$
\$ E * int	+ 5 \$	\$ 3 $\perp$ 2
\$ E	+ 5 \$	\$ 6
\$ E +	5 \$	\$ 6 $\perp$
\$ E + int	\$	\$ 6 $\perp$ 5
\$ E + E	\$	\$ \$ 6 $\perp$ 5
\$ E	\$	\$ 11

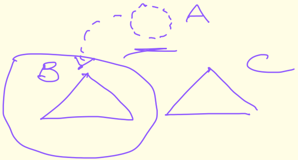
attr. stack

Inherited Attributes and Bottom-up Parsing

- Inherited attributes depend on the *context* in which a symbol is used.
- For inherited attributes, we cannot assign a value to a node's attributes unless the parent's attributes are known.

$A \to B\ C$

- When building parse trees bottom-up, parent of a node is not known when the node is created!



- <u>Solution:</u>
  - Ensure that all attributes are inherited only from left siblings.
  - Use "global" variables to capture inherited values,
  - and introduce "marker" nonterminals to manipulate the global variables.

Inherited Attributes & Bottom-up parsing

Seq of statements

1 statemen

Seq

$$
\begin{array}{lcl}
Ss & \longrightarrow & S\ ;\ Ss \mid \epsilon \\
S & \longrightarrow & B \mid \text{other} \\
B & \longrightarrow & \{\ Ss\ \}
\end{array}
$$

$S \to AaAb$

$S \to BbBa$

$A \to \epsilon$

$B \to \epsilon$

$$
\begin{array}{lcll}
B & \longrightarrow & \{\ M_1\ Ss\ M_2\ \} & \\
M_1 & \longrightarrow & \epsilon & \{current\_block{+}{+};\} \\
M_2 & \longrightarrow & \epsilon & \{current\_block{-};\}
\end{array}
$$

B

{ A Ss }

Ss. nesting_depth

= B. nesting depth

Attribute Grammars

- syntax-directed definitions without side-effects
- attribute definitions can be thought of as *logical assertions* rather than as things that need to be computed
  - distinction between synthesized and inherited attributes disappears

$$
\begin{array}{llll}
E & \longrightarrow & E_1 * E_2 & \{E.type = E_1.type = E_2.type\} \\
E & \longrightarrow & E_1 + E_2 & \{E.type = E_1.type = E_2.type\} \\
E & \longrightarrow & \mathsf{int} & \{E.type = \mathsf{integer}\}
\end{array}
$$

relationships between attributes

Alternatively constraints between attr. values.

Attribute Grammars

An attribute grammar AG is given by (G, V, F), where:

- G is a context-free grammar
- V is the set of attributes, each of which is associated with a terminal or a nonterminal
- F is the set of attribute assertions, each of which is associated with a production in the grammar

A string $s \in L(AG)$ iff $\boxed{s \in L(G)}$ and the attribute assertions hold for production used to derive s, i.e., $\exists$ a parse tree for s w.r.t. G where assertions associated with each edge in the parse tree are satisfied.

Semantic Analysis Phases of Compilation

- Build an Abstract Syntax Tree (AST) while parsing
- Decorate the AST with type information (type checking/inference)
- Generate intermediate code from AST
- Optimize intermediate code
- Generate final code

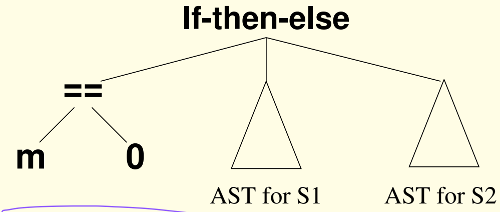
Abstract Syntax Tree (AST)

- Represents syntactic structure of a program
- Abstracts out irrelevant grammar details

An AST for the statement:

“`if (m == 0) S1 else S2`”

is



Abstracted version of Parse Tree

Construction of Abstract Syntax Trees

Typically done simultaneously with parsing

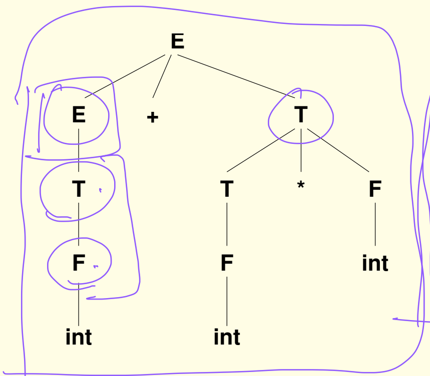
... as another instance of syntax-directed translation

... for translating *concrete* syntax (the parse tree) to *abstract* syntax (AST).

... with AST as a *synthesized attribute* of each grammar symbol.

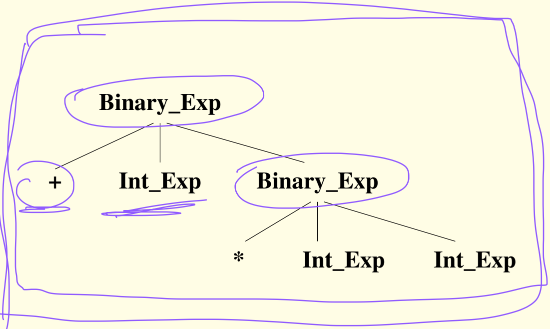
Abstract Syntax Trees

Parse Tree



5 + 2 * 3

AST



`+( 5, *( 2, 3))`

Actions and AST

$$
\begin{array}{lcl}
E & \longrightarrow & E_1 + T \\
 & & \{ E.\texttt{ast} = \texttt{new BinaryExpr(OP\_PLUS,} \\
 & & \qquad\qquad\qquad\qquad E_1.\texttt{ast}, T.\texttt{ast}); \} \\
E & \longrightarrow & T \quad \{ E.\texttt{ast} = T.\texttt{ast}; \} \\
\vdots & & \\
F & \longrightarrow & (E) \quad \{ F.\texttt{ast} = E.\texttt{ast}; \} \\
F & \longrightarrow & \texttt{int} \\
 & & \{ F.\texttt{ast} = \texttt{new IntValNode(int.val);} \}
\end{array}
$$

Actions and AST: Another Example

$$
\begin{array}{lcl}
S & \longrightarrow & \textbf{if } E\ S_1 \textbf{ else } S_2 \\
 & & \{\, S.\texttt{ast} = \texttt{new IfStmtNode(}E.\texttt{ast,} \\
 & & \qquad\qquad\qquad\qquad S_1.\texttt{ast, } S_2.\texttt{ast); }\} \\
S & \longrightarrow & \textbf{return } E \\
 & & \{\, S.\texttt{ast} = \texttt{new ReturnNode(}E.\texttt{ast)}\}
\end{array}
$$

stmt → simplestmt TOK_SEMI

| compoundstmt optSemi

compoundstmt → TOK_LBRACE stmtSeq TOK_RBRACE

optSemi → /*empty*/ | TOK_SEMI

int a[5] = {1, 2, 3, 4, 5};