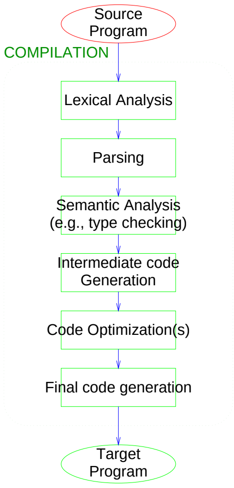
Compilation



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 (*i.e.*, *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*
→ 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

$$\boxed{\begin{array}{lcl} E & \longrightarrow & E * E \\ E & \longrightarrow & E + E \\ E & \longrightarrow & \mathsf{id} \end{array}}$$

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

Syntax-Directed Definitions with yacc

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

$$\begin{array}{lcll} E & : & E\ \mathsf{MULT}\ E & \{\$\$.\mathit{val} = \$1.\mathit{val} * \$3.\mathit{val}\} \\ E & : & E\ \mathsf{PLUS}\ E & \{\$\$.\mathit{val} = \$1.\mathit{val} + \$3.\mathit{val}\} \\ E & : & \mathsf{INT} & \{\$\$.\mathit{val} = \$1.\mathit{val}\} \end{array}$$

Another Example of Syntax-Directed Translation

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

$$\begin{array}{rcll} \mathit{Decl} & \longrightarrow & \mathit{Type}\ \mathit{VarList} & \{\mathit{VarList}.\mathit{type} := \mathit{Type}.\mathit{type}\} \\ \mathit{Type} & \longrightarrow & \ldots & \{\mathit{Type}.\mathit{type} := \ldots\} \\ \mathit{VarList} & \longrightarrow & \mathsf{id}\ ,\ \mathit{VarList}_1 & \{\mathit{VarList}_1.\mathit{type} := \mathit{VarList}.\mathit{type}; \\ & & & \ \ \mathsf{id}.\mathit{type} := \mathit{VarList}.\mathit{type}\} \\ \mathit{VarList} & \longrightarrow & \mathsf{id} & \{\mathsf{id}.\mathit{type} := \mathit{VarList}.\mathit{type}\} \end{array}$$

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:

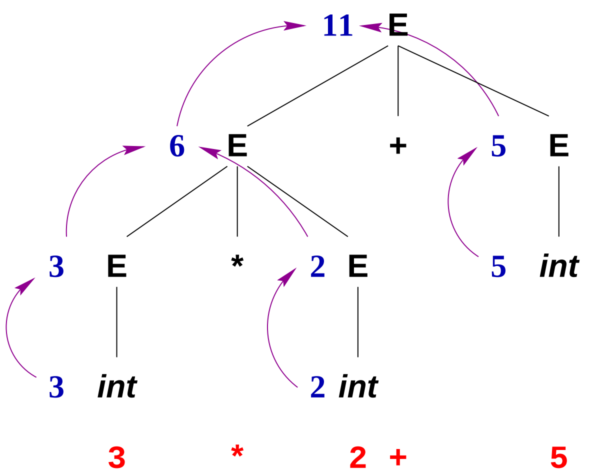
- $A.\mathit{attr} := f(X.\mathit{attr}', Y.\mathit{attr}'')$ for synthesized attributes
- $Y.\mathit{attr} := f(A.\mathit{attr}', X.\mathit{attr}'')$ for inherited attributes

Synthesized Attributes: An Example

$$\boxed{\begin{array}{rcl} E & \longrightarrow & E * E \\ E & \longrightarrow & E + E \\ E & \longrightarrow & \mathsf{int} \end{array}}$$

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

Information Flow for Synthesized Attributes

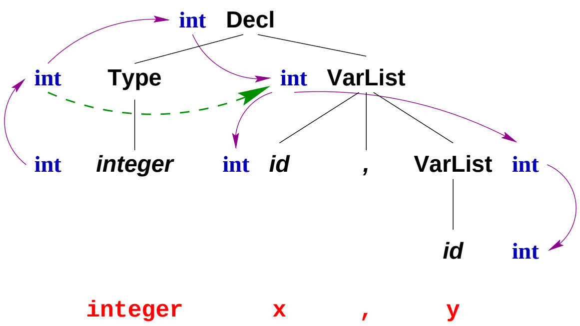


Another Example of Syntax-Directed Translation

Decl	$\longrightarrow$	*Type VarList*
Type	$\longrightarrow$	integer
Type	$\longrightarrow$	float
VarList	$\longrightarrow$	id , *VarList*
VarList	$\longrightarrow$	id

Decl	$\longrightarrow$	*Type VarList*	$\{VarList.type := Type.type\}$
Type	$\longrightarrow$	integer	$\{Type.type := int\}$
Type	$\longrightarrow$	float	$\{Type.type := float\}$
VarList	$\longrightarrow$	id , $VarList_1$	$\{VarList_1.type := VarList.type;$ $\text{id}.type := VarList.type\}$
VarList	$\longrightarrow$	id	$\{\text{id}.type := VarList.type\}$

Information Flow for Inherited Attributes



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

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 & Shift-reduce parsing

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

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

Inherited Attributes and Bottom-up Parsing

Inherited attributes depend on the *context* in which a symbol is used.

For inherited attributes, we cannot assign an value to a node's attributes unless the parent's attributes are known.

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

Solution: 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

$$
\boxed{
\begin{array}{rcl}
Ss & \longrightarrow & S \ ; \ Ss \mid \epsilon \\
S & \longrightarrow & B \mid \mathsf{other} \\
B & \longrightarrow & \{ \ Ss \ \}
\end{array}
}
$$

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

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

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

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 $s \in L(G)$ 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.

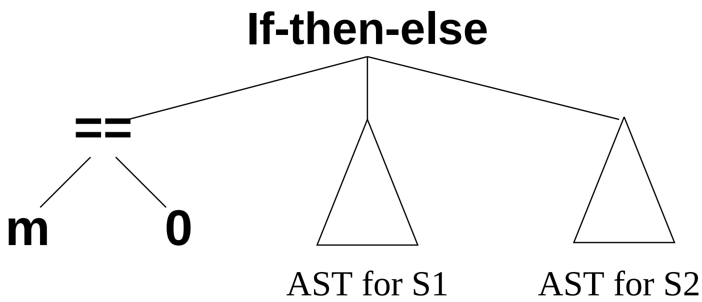
Semantic Analysis Phases of Compilation

- 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



Construction of Abstract Syntax Trees

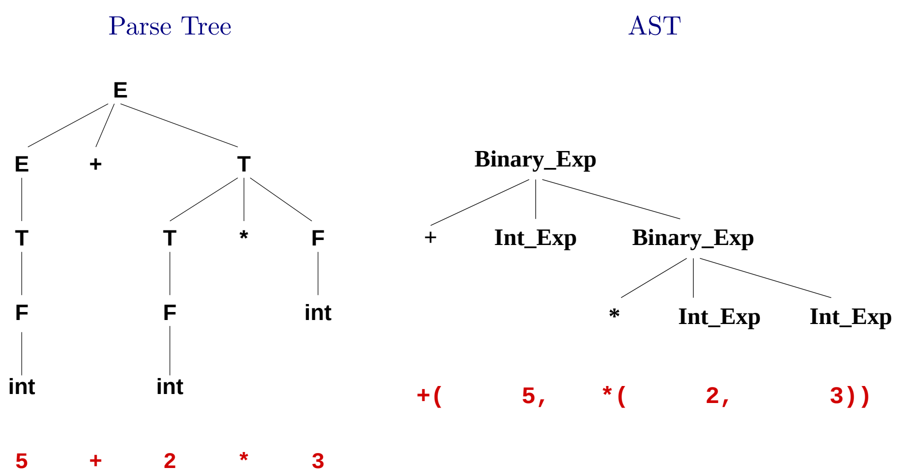
Typically done simultaneously with parsing

... 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



Actions and AST

$E \longrightarrow E_1 + T$
$\{$ E`.ast` $=$ `new BinaryExpr(OP_PLUS,` E_1`.ast,`T`.ast);` $\}$

$E \longrightarrow T \quad \{$ E`.ast` $=$ T`.ast;` $\}$

$\vdots$

$F \longrightarrow (E) \quad \{$ F`.ast` $=$ E`.ast;` $\}$

$F \longrightarrow$ int
$\{$ F`.ast` $=$ `new IntValNode(int.val);` $\}$

Actions and AST: Another Example

$S \longrightarrow$ if E S_1 else S_2
$\{$ S`.ast` $=$ `new IfStmtNode(`E`.ast,` S_1`.ast,` S_2`.ast);` $\}$

$S \longrightarrow$ return E
$\{$ S`.ast` $=$ `new ReturnNode(`E`.ast)`$\}$