Introduction to Compilers

Language Processors and Motivation

Carmen Johana Calderón Chona

About notation

- **Expressing algorithms:**
  - They are written in a slanted, sans-serif font
  - Indentation is both deliberate and significant

```
if Action [s,word] = “shift s_i” then
   push word
   push s_i
   word ← NextWord()
else if ···
```

- **Writting code:** Actual program text is written in a monospace font

```
for i = 1 to n
   read d
   a ← a × 2 × b × c × d
   end
```

- **Arithmetic operators.** Authors have forsaken the traditional use of * for $\times$ and of / for $\div$, except in actual program text.

Programming Languages

- Programming languages are **notations for describing computations**
- Designed for:
  - Human understanding
  - Machine execution
- All software must ultimately be expressed in a form executable by hardware

Why Translation is Needed

- Programs are written in **high-level languages**
- Computers execute **machine-level instructions**
- Translation bridges this gap

What is a Compiler?

Definition

A **compiler** translates a program written in a *source language* into an equivalent program in a *target language*.

- Reports errors in the source program
- Target program is often executable machine code

Example: Source-to-Source Translator

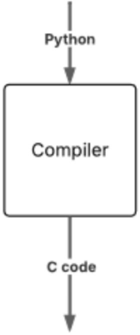


Figure: There exist compilers from source code to source code

Example: Source-to-Source Translators

- Target language is another high-level language
- Example: compilers that emit C code
- Improves portability
- Common in research compilers

Example: PostScript

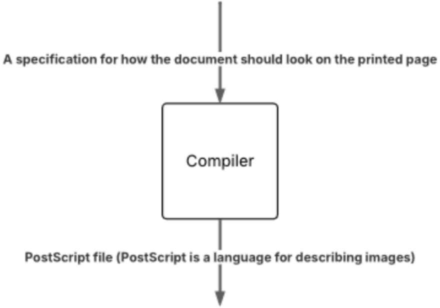


Figure: A typesetting program that produces PostScript

Example: Java compiler

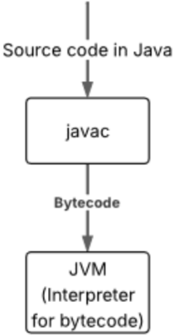


Figure: The Java compiler method (an interesting case)

Why Study Compiler Construction?

- Large and complex software systems
- Strong software engineering challenge
- Design decisions have global impact

Compilers as a Microcosm of CS

- Algorithms:
  - Graph algorithms
  - Dynamic programming
  - Greedy heuristics
- Theory:
  - Automata
  - Formal languages
  - Lattices

Theory Meets Practice

- Scanners and parsers from formal language theory
- Type systems and static analysis
- Code generation and optimization

The fundamentals principles of compilation

- The compiler must preserve the meaning of the program being compiled
- The compiler must improve the input in some discernible way

Other Language Processors

- Interpreters
- Hybrid systems

Compiler vs Interpreter

Compiler

- Produces a target program
- Execution happens later
- Typically faster execution

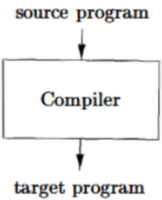


Figure: A compiler

Interpreter

- Executes source program directly
- Statement by statement
- Better runtime error diagnostics

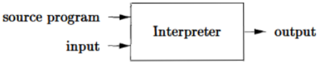


Figure: An interpreter

Hybrid Approaches: Java

- Source code compiled to **bytecode**
- Bytecode executed by a **virtual machine**
- Portable across architectures
- **Just-In-Time (JIT)** compilation for performance

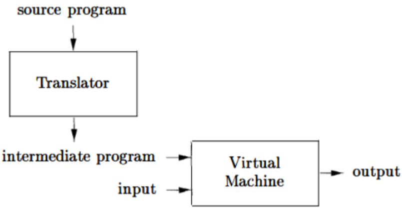


Figure: A hybrid compiler

Other program construction processes

- In addition to a compiler, several other programs may be required to create an executable target program

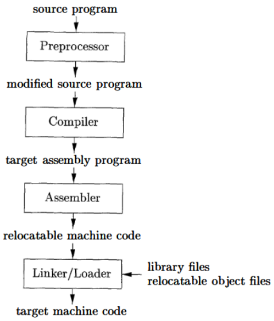


Figure: A language-processing system

Key Takeaway

- Compilers are fundamental to computing
- They combine theory, systems, and engineering
- Studying them builds deep understanding of how computers work

Section 1

The Structure of a Compiler

From Black Box to Structure

- A compiler maps a source program to an equivalent target program
- Internally, this mapping is divided into two major parts:
  - **Analysis**
  - **Synthesis**

Compilation as a Sequence of Phases

- Compilation proceeds in **phases**
- Each phase transforms one representation into another
- The symbol table is shared across phases

Compilation as a Sequence of Phases

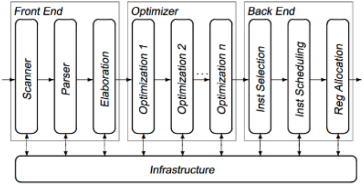


Figure: Structure of a typical compiler

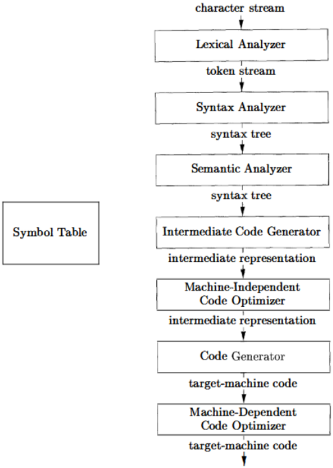


Figure: A more detailed structure

Analysis vs Synthesis

Analysis (Front End)

- Understands the source program
- Checks correctness
- Builds an intermediate representation
- Populates the symbol table

Synthesis (Back End)

- Produces target code
- Uses IR and symbol table
- Maps computation to machine resources

Typical Compiler Phases

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization (optional)
6. Code Generation

Lexical Analysis

- First phase of the compiler
- Groups characters into **lexemes**
- Produces **tokens** for the parser

Token Format

{ token-name, attribute-value }

Example: Lexical Analysis

Source statement:

```
position = initial + rate * 60
```

Token stream:

$$(id, 1)\ (=)\ (id, 2)\ (+)\ (id, 3)\ (*)\ (60)$$

- Blanks are discarded
- Identifiers reference symbol table entries

- Also called **parsing**
- Uses token stream to build a **syntax tree**
- Captures grammatical structure

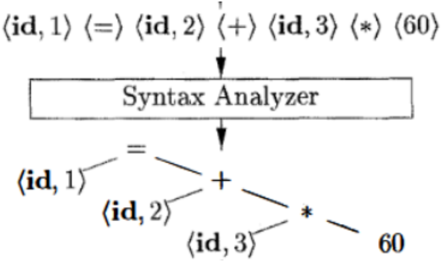


Figure: AST for the above expression

Semantic Analysis

- Uses syntax tree and symbol table
- Gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation
- Checks semantic consistency
- Performs **type checking**

Type Checking and Coercions

- Ensures operators have compatible operands
- May insert implicit type conversions
- Example: integer to floating-point coercion

Intermediate Code Generation

- Produces a low-level, machine-independent IR
- Easy to generate
- Easy to translate into target code

An intermediate representation: Three-Address Code

- Each instruction has at most one operator
- Explicit evaluation order
- Uses temporary variables

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Figure: Some "three-address instructions" like the first and last in this sequence have fewer than three operands

Code Optimization

- Optional phase
- Improves intermediate code
- Common goals:
  - Faster execution
  - Smaller code
  - Lower power consumption

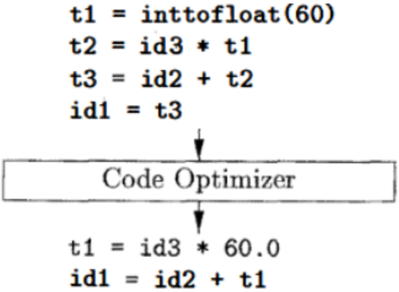


Figure: Illustration of a code optimization

Nature of Optimization

- Optimization problems are rarely solvable optimally
- Heuristic-based techniques dominate
- Improvements, not perfection

Code Generation

- Maps IR to target language
- Selects instructions
- An important issue in code generation: To assign registers and memory locations

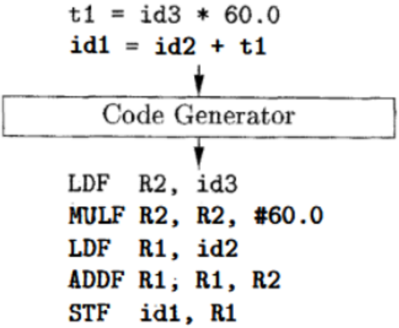


Figure: An example of the action of a code generator

Key Challenge: Register Allocation

- The organization of storage at run-time depends on the language being compiled.
  - Each language has different rules. For example, in C, there is explicit use of the stack and heap, and manual memory allocation, while functional languages involve a lot of object creation and closures. **Therefore, the compiler cannot handle memory the same way for all languages**
- Storage-allocation decisions are made either during intermediate code generation or during code generation
- Registers are limited
- Poor allocation leads to slow code
- Central problem in back-end design

Symbol-Table Management

- Stores information about identifiers
- Attributes include:
  - Type
  - Scope
  - Storage location
  - Procedure signatures

1	position	...
2	initial	...
3	rate	...

Figure: Symbol table

Phases vs Passes

- **Phase**: logical organization
- **Pass**: implementation-level traversal
- Multiple phases may be grouped into one pass

Compiler Construction Tools

- Scanner generators
- Parser generators
- Syntax-directed translation tools
- Code-generator generators
- Data-flow analysis engines

Evolution of Programming Languages (I)

- **1940s: Machine Languages**
  - Programs written in 0s and 1s
  - Very low-level, error-prone, hard to maintain
- **1950s: Assembly Languages**
  - Mnemonic instructions
  - Macros for reusable instruction sequences
- **Rise of High-Level Languages**
  - Fortran (scientific computing)
  - Cobol (business applications)
  - Lisp (symbolic computation)

Evolution of Programming Languages (II)

- **Generational Classification**
  - 1GL: Machine 2GL: Assembly 3GL: High-level
  - 4GL: Domain-specific (SQL, PostScript)
  - 5GL: Logic and constraint-based (Prolog)
- **Programming Paradigms**
  - Imperative: C, C++, Java
  - Declarative: Haskell, Prolog
- **Other Classifications**
  - Von Neumann languages (e.g., C, Fortran)
  - Object-Oriented languages (C++, Java, Ruby)
  - Scripting languages (Python, JavaScript, Perl)

The Science of Building a Compiler

- **Abstraction and Modeling**
  - Real-world language problems solved using mathematical models
  - Key models: finite automata, regular expressions, grammars, trees
- **Correctness and Scale**
  - Compilers must handle infinitely many valid programs
  - All transformations must preserve program meaning
- **Code Optimization as a Science**
  - Optimization improves performance, not guaranteed optimality
  - Based on rigorous theory (graphs, data-flow, linear models)
  - Validated through experimentation
- **Design Objectives**
  - Correctness (most critical)
  - Performance improvement
  - Reasonable compilation time
  - Manageable engineering complexity

Applications of Compiler Technology

- **Implementation of High-Level Languages**
  - Translation of abstractions into efficient machine code
  - Optimizations: register allocation, data-flow analysis, inlining
- **Architectural Optimization**
  - Exploiting parallelism (ILP, multithreading)
  - Managing memory hierarchies (registers, caches)
- **Computer Architecture Design**
  - Influence on RISC, VLIW, SIMD architectures
  - Compiler-driven architectural evaluation
- **Program Translation**
  - Binary translation and backward compatibility
  - Hardware synthesis (Verilog, VHDL)
  - Database queries and compiled simulation
- **Software Productivity and Security**
  - Static analysis, type checking, bounds checking
  - Detection of bugs and security vulnerabilities

Section 2

Introducing some important concepts with a Simple Syntax-Directed Translator

Things to remember

Analysis phase

- Breaks up a source program into constituent pieces and produces an internal representation for it, called intermediate code
- Analysis is organized around the "syntax" of the language to be compiled
- The syntax of a programming language describes the proper form of its programs

Synthesis phase

- Translates the intermediate code into the target program

Things to remember

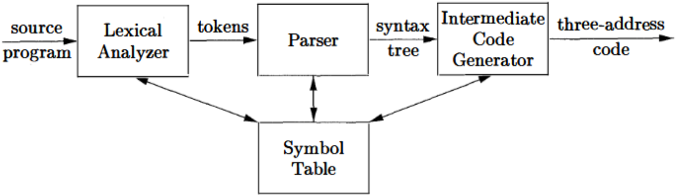
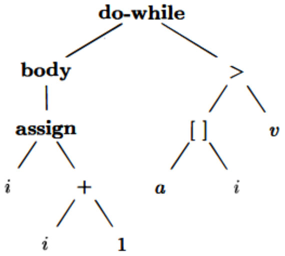


Figure: A model of a compiler front end

***Some compilers combine parsing and intermediate-code generation into one component.



(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

Figure 2.4: Intermediate code for "`do i=i+1; while(a[i]<v);`"

Context-free grammars

A *context-free grammar* G is defined by the 4-tuple

$$G = (V, \Sigma, R, S),$$

where:

- V is a finite set. Each element $v \in V$ is called a *nonterminal* (or *variable*). Each variable represents a different type of phrase or clause in a sentence. Variables are also called *syntactic categories*. Each variable defines a sublanguage of the language generated by G.
- Σ is a finite set of *terminal symbols*, disjoint from V, which form the actual content of the sentences. The set Σ is the alphabet of the language generated by the grammar G.
- R is a finite relation

  $$R \subseteq V \times (V \cup \Sigma)^*,$$

  where * denotes the Kleene star operation. The elements of R are called *(rewrite) rules* or *productions* (often denoted by P).
- $S \in V$ is the *start symbol*, which represents the entire sentence (or program).

Example: Context-Free Grammar for Arithmetic Expressions

We consider expressions consisting of digits separated by plus or minus signs, such as:

$$9 - 5 + 2, \quad 3 - 1, \quad 7$$

Such expressions are called *lists of digits separated by plus or minus signs*. The following context-free grammar describes their syntax.

Productions:

$$\begin{aligned} \mathit{list} &\rightarrow \mathit{list} + \mathit{digit} \\ \mathit{list} &\rightarrow \mathit{list} - \mathit{digit} \\ \mathit{list} &\rightarrow \mathit{digit} \\ \mathit{digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$

Terminals: $\{+, -, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Nonterminals: *list*, *digit*
Start symbol: *list*

Derivations in Context-Free Grammars

A grammar derives strings by:

- Starting from the *start symbol*,
- Repeatedly replacing a nonterminal with the body of one of its productions.

The set of all *terminal strings* that can be derived from the start symbol forms the *language* defined by the grammar.

Example (Grammar from Example 2.1):

- The language consists of lists of digits separated by plus and minus signs.
- The nonterminal *digit* can generate any digit from 0 to 9.
- A single digit is a valid list.
- Any list followed by $+$ or $-$ and another digit is also a list.

Example of a Derivation

We show that the string $9 - 5 + 2$ belongs to the language.

1. 9 is a list, since 9 is a digit (by production $\mathit{list} \rightarrow \mathit{digit}$)
2. $9 - 5$ is a list, since 9 is a list and 5 is a digit (by production $\mathit{list} \rightarrow \mathit{list} - \mathit{digit}$)
3. $9 - 5 + 2$ is a list, since $9 - 5$ is a list and 2 is a digit (by production $\mathit{list} \rightarrow \mathit{list} + \mathit{digit}$)

Thus, $9 - 5 + 2$ is derived from the start symbol *list*.

Parsing

Parsing is the problem of:

- Taking a string of terminals, and
- Determining how it can be derived from the start symbol of a grammar.

If the string *cannot* be derived from the start symbol, the parser reports *syntax errors* in the string.

Parsing is one of the most fundamental problems in compiling. The main parsing techniques are studied later.

Parsing and Lexical Analysis

For simplicity, we begin with examples such as:

$$9 - 5 + 2$$

where each character is treated as a terminal symbol.

In general:

- Source programs consist of *multicharacter lexemes*.
- A *lexical analyzer* groups lexemes into *tokens*.
- The first component of each token is a terminal symbol processed by the parser.

Parse Trees

A **parse tree** pictorially shows how the start symbol of a grammar derives a string in the language.

If a nonterminal A has a production

$$A \to XYZ,$$

then the parse tree may contain an interior node labeled A with three children, labeled X, Y, and Z from left to right.

$$\begin{array}{ccc} & A & \\ / & | & \backslash \\ X & Y & Z \end{array}$$

Formal Definition of a Parse Tree

Given a context-free grammar, a **parse tree** satisfies:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by ε.
3. Each interior node is labeled by a nonterminal.
4. If an interior node labeled A has children labeled $X_1, X_2, \ldots, X_n$ (from left to right), then there must be a production

$$A \to X_1 X_2 \cdots X_n.$$

As a special case, if $A \to \varepsilon$ is a production, then a node labeled A may have a single child labeled ε.

Example: Parse Tree for 9−5+2

The derivation of the string $9 - 5 + 2$ can be illustrated using a **parse tree**.

Each node in the tree is labeled by a grammar symbol:

- An *interior node* corresponds to the head of a production.
- Its *children* correspond to the body of that production.

The root of the tree is labeled *list*, which is the start symbol of the grammar in Example 2.1.

The children of the root, from left to right, are:

$$\mathit{list} \quad + \quad \mathit{digit}$$

corresponding to the production:

$$\mathit{list} \rightarrow \mathit{list} + \mathit{digit}.$$

Parse Tree for 9−5+2

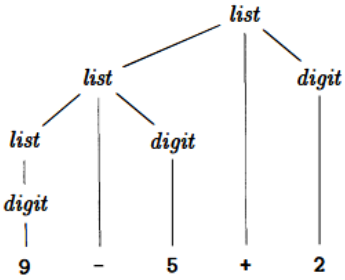


Figure 2.5: Parse tree for 9−5+2 according to the grammar in Example 2.1

Ambiguity in Context-Free Grammars

- A grammar is said to be **ambiguous** if there exists a terminal string that can be generated by *more than one parse tree*.
- To show that a grammar is ambiguous, it suffices to find a single string that has multiple parse trees.
- Since different parse trees usually correspond to different meanings, **unambiguous grammars** are preferred in compiling. Alternatively, ambiguous grammars may be used together with additional rules to resolve ambiguities.

Example 2.5

Example: Consider the grammar with a single nonterminal:

$$string \to string + string \mid string - string \mid 0 \mid 1 \mid \cdots \mid 9$$

With this grammar, the expression $9 - 5 + 2$ has more than one parse tree, corresponding to:

$$(9 - 5) + 2 \quad \text{and} \quad 9 - (5 + 2).$$

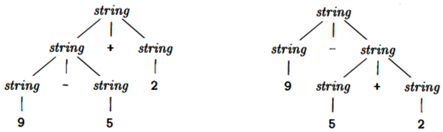


Figure 2.6: Two parse trees for 9-5+2

Associativity of Operators

When an operand has operators on both sides, **associativity** determines which operator applies first.

Left-associative operators:

$$9 + 5 + 2 = (9 + 5) + 2$$

$$9 - 5 - 2 = (9 - 5) - 2$$

Most arithmetic operators in programming languages ($+$, $-$, $*$, $/$) are *left-associative*.

Right-Associative Operators

Some operators associate to the **right**.

Example (assignment in C-like languages):

$$a = b = c \quad \equiv \quad a = (b = c)$$

A grammar for a right-associative operator:

$$\mathit{right} \rightarrow \mathit{letter} = \mathit{right} \mid \mathit{letter}$$

$$\mathit{letter} \rightarrow a \mid b \mid \cdots \mid z$$

Parse trees:

- Left-associative trees grow to the left
- Right-associative trees grow to the right

Precedence of Operators

Associativity applies to *the same operator*. When different operators appear, **precedence** is needed.

Example:

$$9 + 5 * 2$$

Two interpretations:

$$(9 + 5) * 2 \quad \text{or} \quad 9 + (5 * 2)$$

In arithmetic:

- $*$ and $/$ have higher precedence than $+$ and $-$
- Therefore: $\quad 9 + (5 * 2)$

Grammar with Precedence and Associativity

We encode precedence using different nonterminals:

$$expr \rightarrow expr + term \mid expr - term \mid term$$

$$term \rightarrow term * factor \mid term / factor \mid factor$$

$$factor \rightarrow digit \mid (\, expr \,)$$

Higher-precedence operators appear lower in the grammar.

Syntax-Directed Translation

Syntax-directed translation attaches semantic actions (program fragments) to the productions of a grammar.

Translation exploits the *structure* revealed by parsing.

Example production:

$$expr \rightarrow expr_1 + term$$

Intuition:

- Translate the left subexpression
- Translate the right subexpression
- Then handle the operator

Translation Guided by Structure

For the production:

$$expr \rightarrow expr_1 + term$$

A structure-based translation follows the parse tree:

```
translate(expr_1);
translate(term);
handle(+);
```

Later, this idea will be used to:

- Build syntax trees
- Evaluate expressions
- Translate infix to postfix notation

Attributes and Translation Schemes

Attributes are quantities associated with grammar symbols.

Examples:

- Value of an expression
- Data type
- Generated code or instruction count

A **translation scheme**:

- Attaches program fragments to grammar productions
- Executes them during syntax analysis
- Produces the translation as a combined result

Postfix Notation

Postfix notation places operators *after* their operands.

Defined inductively:

1. A variable or constant translates to itself
2. E_1 *op* E_2 translates to E_1' E_2' *op*
3. Parentheses do not change the translation

Postfix notation requires no parentheses and is unambiguous.

Postfix Notation: Examples

$$(9-5)+2 \quad \Rightarrow \quad 95-2+$$

Steps:

- $9-5 \Rightarrow 95-$
- $(9-5)$ stays the same
- Combine with $+2 \Rightarrow 95-2+$

Another example:

$$9-(5+2) \Rightarrow 952+-$$

Synthesized Attributes

Attributes can be computed *from the children to the parent* in a parse tree.

A **syntax-directed definition** specifies:

- Attributes for grammar symbols
- Semantic rules for each production

For an input string:

1. Construct the parse tree
2. Evaluate attributes using semantic rules

A parse tree with attribute values is called an **annotated parse tree**.

Synthesized and Inherited Attributes

An attribute is said to be **synthesized** if its value at a parse-tree node is determined only from:

- Attribute values of its children, and
- Information at the node itself.

Key property: Synthesized attributes can be evaluated in a *single bottom-up traversal* of the parse tree.

Another important kind of attribute is the **inherited attribute**:

- Its value depends on the node itself,
- Its parent, and
- Its siblings in the parse tree.

Inherited attributes are discussed later (Section 5.1.1).

Example: Synthesized Attribute for Postfix Translation

Each nonterminal has a synthesized attribute t, representing the postfix notation of the generated expression.

Basic rules:

- A digit translates to itself:

$$term \to 9 \quad \Rightarrow \quad term.t = \texttt{"9"}$$

- If $expr \to term$, then:

$$expr.t = term.t$$

Plus operator:

$$expr \to expr_1 + term$$

Here, $||$ denotes string concatenation.

$$expr.t = expr_1.t \;||\; term.t \;||\; \texttt{"+"}$$

PRODUCTION	SEMANTIC RULES
$expr \to expr_1 + term$	$expr.t = expr_1.t \;\|\|\; term.t \;\|\|\; '+'$
$expr \to expr_1 - term$	$expr.t = expr_1.t \;\|\|\; term.t \;\|\|\; '-'$
$expr \to term$	$expr.t = term.t$
$term \to 0$	$term.t = '0'$
$term \to 1$	$term.t = '1'$
$\cdots$	$\cdots$
$term \to 9$	$term.t = '9'$

Figure 2.10: Syntax-directed definition for infix to postfix translation

Simple Syntax-Directed Definitions

A syntax-directed definition is called **simple** if:

- The translation of the head nonterminal is formed by
- Concatenating the translations of the nonterminals in the body,
- In the *same order* as they appear in the production,
- With optional additional strings interleaved.

This property holds for the syntax-directed definition used to translate infix expressions into postfix notation.

Example: Simple Syntax-Directed Definition

Production:

$$expr \rightarrow expr_1 + term$$

Semantic rule:

$$expr.t = expr_1.t \,||\, term.t \,||\, \texttt{"+"}$$

Key observations:

- The translations of $expr_1$ and *term* appear in the same order as in the production.
- No symbols are inserted before or between them.
- The only additional symbol (+) appears at the end.

Simple definitions can be implemented by printing only the additional strings, in the order they appear.

Tree Traversals

Tree traversals are used to:

- Describe how attributes are evaluated, and
- Specify when code fragments are executed in translation schemes.

A traversal:

- Starts at the root, and
- Visits every node of the tree in some order.

A **depth-first traversal**:

- Visits a node, then recursively visits its children,
- Goes as deep as possible before moving to other nodes.

Synthesized attributes can be evaluated using a **bottom-up** traversal (i.e., after visiting all children).

Preorder and Postorder Traversals

In depth-first traversals, children are typically visited from left to right.

Preorder traversal:

- Action is performed when the node is first visited.
- Order: Node $\rightarrow$ Children

Postorder traversal:

- Action is performed after all children have been visited.
- Order: Children $\rightarrow$ Node

Postorder traversals are especially important because:

- They naturally support evaluation of synthesized attributes.

From SDDs to Translation Schemes

- Syntax-Directed Definitions (SDDs) build translations using attributes.
- Attributes often store strings attached to parse-tree nodes.
- Translation Schemes provide an alternative:
  - No string manipulation
  - Translation produced incrementally

Translation Scheme

Definition:

A *syntax-directed translation scheme* specifies a translation by:

- Attaching **program fragments** to grammar productions
- Explicitly defining the **order of execution**

Semantic rules are written directly inside the productions.

Semantic Actions

- Program fragments embedded in production bodies
- Enclosed in curly braces $\{\ \}$
- Executed when their position in the production is reached

Example:

$$\text{rest} \rightarrow + \text{ term } \{\text{print}('+')\} \text{ rest}_1$$

Parse Trees with Actions

- Semantic actions are represented as extra nodes
- Connected by dashed lines to the head of the production
- Action nodes have no children

Execution rule:

- Action is performed when its node is first visited

Postorder Traversal

- Translation schemes are executed as if:
  - A parse tree were built
  - Semantic actions were executed during a **postorder traversal**
- Actual parse tree construction is not required

Infix to Postfix Translation

Input expression:

$$9 - 5 + 2$$

Postfix output:

$$95 - 2+$$

- Digits are printed immediately
- Operators are printed after both operands

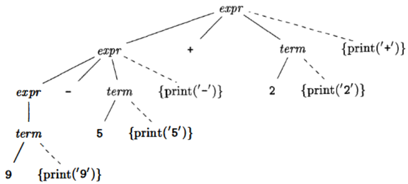


Figure 2.14: Actions translating 9-5+2 into 95-2+

expr	→	$expr_1$ + *term*	{print('+')}
expr	→	$expr_1$ - *term*	{print('-')}
expr	→	*term*	
term	→	0	{print('0')}
term	→	1	{print('1')}
		...	
term	→	9	{print('9')}

Figure 2.15: Actions for translating into postfix notation

Why It Works

- Postorder traversal ensures:
  - Left operand processed first
  - Right operand processed next
  - Operator printed last
- Each character is printed exactly once
- No storage for intermediate results is required

SDD vs Translation Scheme

SDD	**Translation Scheme**
Uses attributes	Uses semantic actions
Builds strings	Prints output incrementally
Order implicit	Order explicit
Declarative	Operational