Compilers

Nathan Warner

Northern Illinois
University

Computer Science
Northern Illinois University
United States

Contents

Introduction

- **Compilers**: In its most general form, a compiler is a program that accepts as input a program text in a certain language and produces as output a program text in another language, while preserving the meaning of that text

- **Translation, source language, target language, and implementation language**: This process is called translation, as it would be if the texts were in natural languages. Almost all compilers translate from one input language, the source language, to one output language, the target language, only. One normally expects the source and target language to differ greatly: the source language could be C and the target language might be machine code for the Pentium processor series. The language the compiler itself is written in is the implementation language.

  To obtain the translated program, we run a compiler, which is just another program whose input is a file with the format of a program source text and whose output is a file with the format of executable code

- **Bootstrapping**: When the source language is also the implementation language and the source text to be compiled is actually a new version of the compiler itself, the process is called bootstrapping.

- **Front and back end**: The part of a compiler that performs the analysis of the source language text is called the front-end, and the part that does the target language synthesis is the back-end

  If the compiler has a very clean design, the front-end is totally unaware of the target language and the back-end is totally unaware of the source language, the only thing they have in common is knowledge of the semantic representation

- **Parse tree / syntax tree**: The **syntax tree** of a program text is a data structure which shows precisely how the various segments of the program text are to be viewed in terms of the grammar. The syntax tree can be obtained through a process called "parsing"

  Parsing is the process of structuring a text according to a given grammar. For this reason, syntax trees are also called **parse trees**; we will use the terms interchangeably, with a slight preference for "parse tree" when the emphasis is on the actual parsing. Conversely, parsing is also called syntax analysis

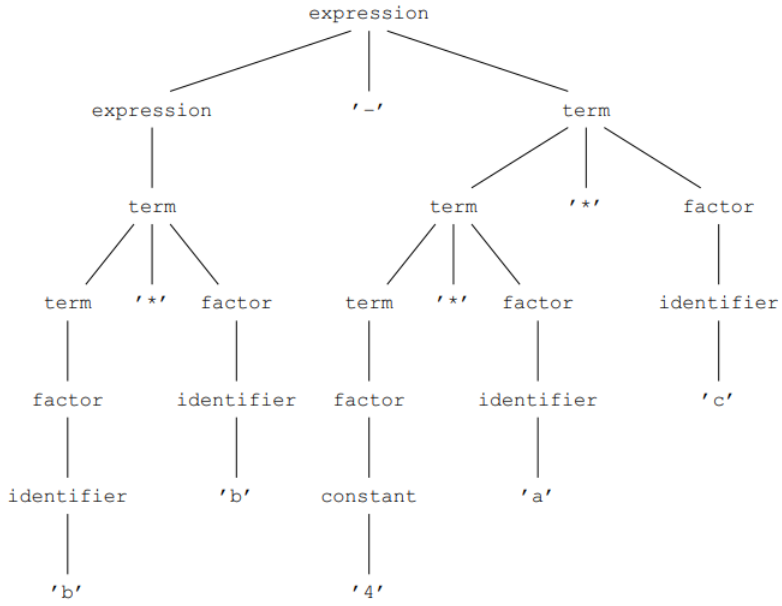
- **Abstract syntax tree (AST)**: The exact form of the parse tree as required by the grammar is often not the most convenient one for further processing, so usually a modified form of it is used, called an abstract syntax tree, or AST. Detailed information about the semantics can be attached to the nodes in this tree through annotations, which are stored in additional data fields in the nodes; hence the term annotated abstract syntax tree. Since unannotated ASTs are of limited use, ASTs are always more or less annotated in practice, and the abbreviation "AST" is used also for annotated ASTs.

- **Lexical analysis**: Usually the grammar of a programming language is not specified in terms of input characters but of input "tokens". Input tokens may be and sometimes must be separated by white space, which is otherwise ignored. So before feeding the input program text to the parser, it must be divided into tokens. Doing so is the task of the lexical analyzer; the activity itself is sometimes called "to tokenize", but the literary value of that word is doubtful.

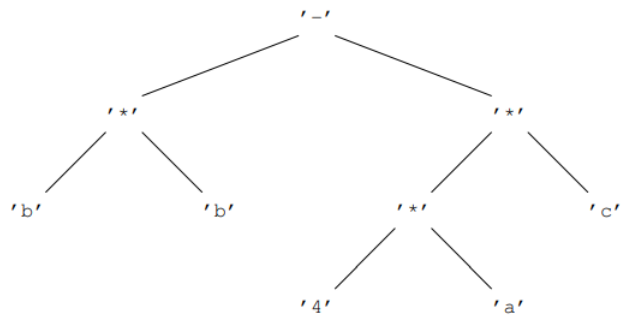
- **Example grammer, parse tree, and AST**: Consider the grammer

$$E \to E + T \mid E - T \mid T$$
$$T \to T * F \mid T/F \mid F$$
$$F \to \text{identifier} \mid \text{constant} \mid (E),$$

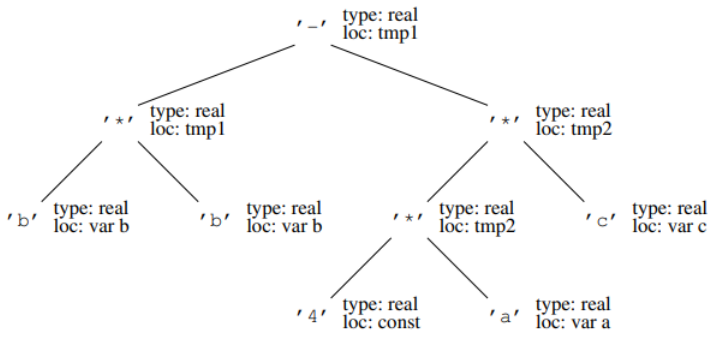
where E is an expression, T is a term, and F is a factor. The parse tree for the expression $b * b - 4 * a * c$ would look something like



Whereas the AST would be



The annotated AST would be



- **Narrow and broad compilers**: A narrow compiler reads a small part of the program, typically a few tokens, processes the information obtained, produces a few bytes of object code if appropriate, discards most of the information about these tokens, and repeats this process until the end of the program text is reached.

  A broad compiler reads the entire program and applies a series of transformations to it (lexical, syntactic, contextual, optimizing, code generating, etc.), which eventually result in the desired object code. This object code is then generally written to a file

- **N-pass compilers**: Since the "field of vision" of a narrow compiler is, well, narrow, it is possible that it cannot manage all its transformations on the fly. Such compilers then write a partially transformed version of the program to disk and, often using a different program, continue with a second pass; occasionally even more passes are used. Not surprisingly, such a compiler is called a 2-pass (or N-pass) compiler, or a 2-scan (N-scan) compiler. If a distinction between these two terms is made, "2-scan" often indicates that the second pass actually re-reads (re-scans) the original program text, the difference being that it is now armed with information extracted during the first scan.

- **Portable programs**: A program is considered portable if it takes a limited and reasonable effort to make it run on different machine types. What constitutes "a limited and reasonable effort" is, of course, a matter of opinion, but today many programs can be ported by just editing the makefile to reflect the local situation and recompiling.

- **Grammers (CFG's)**: Grammars, or more precisely context-free grammars, are the essential formalism for describing the structure of programs in a programming language. In principle the grammar of a language describes the syntactic structure only, but since the semantics of a language is defined in terms of the syntax, the grammar is also instrumental in the definition of the semantics

  There are other grammar types besides context-free grammars, but we will be mainly concerned with context-free grammars. We will also meet regular grammars, which more often go by the name of "regular expressions" and which result from a severe restriction on the context-free grammars; and attribute grammars, which are context-free grammars extended with parameters and code. Other types of grammars play only a marginal role in compiler construction. The term "contextfree" is often abbreviated to CF. We will give here a brief summary of the features of CF grammars

  A "grammar" is a recipe for constructing elements of a set of strings of symbols. When applied to programming languages, the symbols are the tokens in the language, the strings of symbols are program texts, and the set of strings of symbols is the programming language. The string

BEGIN print ("Hi!") END

consists of 6 symbols (tokens) and could be an element of the set of strings of symbols generated by a programming language grammar, or in more normal words, be a program in some programming language. This cut-and-dried view of a programming language would be useless but for the fact that the strings are constructed in a structured fashion; and to this structure semantics can be attached.

the six tokens produced by a lexical analyzer are:

- **BEGIN:** keyword
- **print:** identifier (or keyword, depending on the language specification)
- **(:** left parenthesis
- **"Hi!":** string literal
- **):** right parenthesis
- **END:** keyword

- **The form of a grammar**: A **grammar** consists of a set of production rules and a start symbol. Each production rule defines a named syntactic construct. A **production rule** consists of two parts, a left-hand side and a right-hand side, separated by a left-to-right arrow. The **left-hand side** is the name of the syntactic construct; the **right-hand side** shows a possible form of the syntactic construct. An example of a production rule is

$$\text{expression } \rightarrow' (\text{'expression operator expression})'.$$

- **Terminal and non-terminal symbols**: The right-hand side of a production rule can contain two kinds of symbols, terminal symbols and non-terminal symbols. As the word says, a **terminal symbol** (or **terminal** for short) is an end point of the production process, and can be part of the strings produced by the grammar. A **non-terminal symbol** (or **non-terminal** for short) must occur as the left-hand side (the name) of one or more production rules, and cannot be part of the strings produced by the grammar. Terminals are also called **tokens**, especially when they are part of an input to be analyzed. Non-terminals and terminals together are called **grammar symbols**. The grammar symbols in the righthand side of a rule are collectively called its **members**; when they occur as nodes in a syntax tree they are more often called its "children"
  - **Non-terminals** are denoted by capital letters, mostly A, B, C, and N.
  - **Terminals** are denoted by lower-case letters near the end of the alphabet, mostly x, y, and z.
  - **Sequences of grammar symbols** are denoted by Greek letters near the beginning of the alphabet, mostly α (alpha), β (beta), and γ (gamma).
  - **Lower-case letters near the beginning of the alphabet** (a, b, c, etc.) stand for themselves, as terminals.
  - **The empty sequence** is denoted by ε (epsilon).

- **Sentential form, production tree,and production step**: The central data structure in the production process is the sentential form. It is usually described as a string of grammar symbols, and can then be thought of as representing a partially produced program text. For our purposes, however, we want to represent the syntactic structure of the program too. The syntactic structure can be added to the flat interpretation of a sentential form as a tree positioned above the sentential form so that the leaves of the tree are the grammar symbols. This combination is also called a production tree

A string of terminals can be produced from a grammar by applying so-called production steps to a sentential form, as follows. The sentential form is initialized to a copy of the start symbol. Each production step finds a non-terminal N in the leaves of the sentential form, finds a production rule $N \to \alpha$ with N with N as its lefthand side, and replaces the N in the sentential form with a tree having N as the root and the right-hand side of the production rule, α, as the leaf or leaves. When no more non-terminals can be found in the leaves of the sentential form, the production process is finished, and the leaves form a string of terminals in accordance with the grammar.

Using the conventions described above, we can write that the production process replaces the sentential form $\beta N \gamma$ by $\beta \alpha \gamma$

- **More on sentential forms and production steps**: A production step is a single application of a grammar rule. If a grammar has a rule

$$N \to \alpha$$

then a sentential of the form

$$\beta N \gamma$$

can be written as

$$\beta \alpha \gamma.$$

  - N: A non-terminal to be expanded
  - α: The replacement string
  - β, γ: Unchanged context

When we write

$$\beta N \gamma \implies \beta \alpha \gamma$$

we are describing one production step

  - N — a non-terminal that we are going to expand
  - α — the right-hand side of a grammar rule (what replaces N)
  - β — everything to the left of N
  - γ — everything to the right of N

- **Derivation**: The steps in the production process leading from the start symbol to a string of terminals are called the derivation of that string. Suppose our grammar consists of the four numbered production rules:

  1. expression → '(' expression operator expression ')'
  2. expression → '1'
  3. operator → '+'
  4. operator → ' *'

in which the terminal symbols are surrounded by apostrophes and the non-terminals are identifiers, and suppose the start symbol is expression. Then the sequence of sentential forms shown below

```
      expression
1@1   '(' expression operator expression ')'
2@2   '(' '1' operator expression ')'
4@3   '(' '1' '*' expression ')'
1@4   '(' '1' '*' '(' expression operator expression ')' ')'
2@5   '(' '1' '*' '(' '1' operator expression ')' ')'
3@6   '(' '1' '*' '(' '1' '+' expression ')' ')'
2@7   '(' '1' '*' '(' '1' '+' '1' ')' ')'
```

Fig. 1.26: Leftmost derivation of the string (1*(1+1))

forms the derivation of the string $(1 * (1 + 1))$. More in particular, it forms a **leftmost derivation**, a derivation in which it is always the leftmost non-terminal in the sentential form that is rewritten

An indication $R@P$ shows that grammar rule R is used to rewrite the non-terminal at position P. The resulting parse tree is

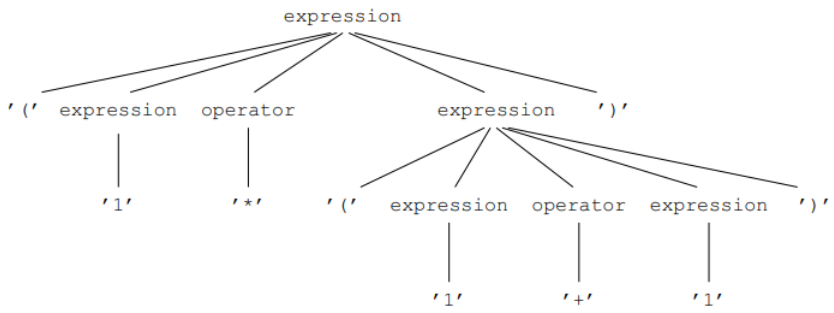


Fig. 1.27: Parse tree of the derivation in Figure 1.26

We see that recursion—the ability of a production rule to refer directly or indirectly to itself—is essential to the production process; without recursion, a grammar would produce only a finite set of strings

The production process is kind enough to produce the program text together with the production tree, but then the program text is committed to a linear medium (paper, computer file) and the production tree gets stripped off in the process. Since we need the tree to find out the semantics of the program, we use a special program, called a "parser", to retrieve it

- **Extended forms of grammars**: The single grammar rule format

  non-terminal $\rightarrow$ zero or more grammar symbols

  used above is sufficient in principle to specify any grammar, but in practice a richer notation is used

  The format described so far is known as BNF, which may be considered an abbreviation of Backus–Naur Form or of Backus Normal Form. It is very suitable for expressing nesting and recursion, but less convenient for expressing repetition and optionality, although it can of course express repetition through recursion. To remedy this, three additional notations are introduced, each in the form of a postfix operator:

- R^+: Indicates the occurrence of one or more Rs, to express repetition
- $R^?$: Indicates the occurrence of zero or one Rs, to express optionality
- R^*: Indicates the occurrence of zero or more Rs, to express optional repetition

Parentheses may be needed if these postfix operators are to operate on more than one grammar symbol. The grammar notation that allows the above forms is called EBNF, for Extended BNF. An example is the grammar rule

$$\text{paramater\_list} \rightarrow (\text{'IN'} \mid \text{'OUT'})^? \text{ identitfier } (\text{','} \text{ identifier})^*.$$

- **Properties of grammars**:
  - **Left-recursive non-terminal**: A non-terminal N is left-recursive if, starting with a sentential form N, we can produce another sentential form starting with N
  - **Left-recursive grammar**: By extension, a grammar that contains one or more left-recursive rules is itself called left-recursive
  - **Right-recursive**: The right version of left-recursive, not as important.
  - **Nullable non-terminal**: A non-terminal N is nullable if, starting with a sentential form N, we can produce an empty sentential form ϵ

    A grammar rule for a nullable non-terminal is called an ϵ-rule.
  - **Useless non-terminal**: A non-terminal N is useless if it can never produce a string of terminal symbols: any attempt to do so inevitably leads to a sentential that again contains N.
  - **Ambiguous grammar**: A grammar is ambiguous if it can produce two different production trees with the same leaves in the same order.

    That means that when we lose the production tree due to linearization of the program text we cannot reconstruct it unambiguously; and since the semantics derives from the production tree, we lose the semantics as well. So ambiguous grammars are to be avoided in the specification of programming languages, where attached semantics plays an important role
- **Symbols**: The basic unit in formal grammars is the symbol. The only property of these symbols is that we can take two of them and compare them to see if they are the same. In this they are comparable to the values of an enumeration type. Like these, symbols are written as identifiers, or, in mathematical texts, as single letters, possibly with subscripts. Examples of symbols are N, x, procedure_body, assignment_symbol, t_k.
- **Production rule**. Given two sets of symbols V_1 and V_2, a production rule is a pair

$$(N, \alpha) \text{ such that } N \in V_1, \alpha \in V_2^*,$$

in which X^* means a sequence of zero or more elements of the set X. This means that a production rule is a pair consisting of an N, which is an element of V_1, and a sequence α of elements of V_2. We call N the *left-hand side* and α the *right-hand side*. We do not normally write this as a pair (N, α), but rather as

$$N \rightarrow \alpha,$$

although technically it is a pair. The V in V_1 and V_2 stands for *vocabulary*.

- **CFGs**: A context-free grammar G is a 4-tuple

$$G = (V_N, V_T, S, P)$$

where V_N and V_T are sets of symbols. S is a symbol, and P is a set of production rules. The elements of V_N are the **non-terminal symbols**, and elements of V_T are the **terminal symbols**. S is called the **start symbol**.

- **Context conditions**: The previous paragraph defines only the context-free form of a grammar. To make it a real, acceptable grammar, it has to fulfill three context condition

1. $V_N \cap V_T = \varnothing$
2. $S \in V_N$
3. $P \subseteq \{(N, \alpha) \mid N \in V_N, \alpha \in (V_N \cup V_T)^*$

- **Strings**: Sequences of symbols are called strings.

- **Directly derivable**: A string may be derivable from another string in a grammar. More precisely, a string β is said to be *directly derivable* from a string α, written as

$$\alpha \Rightarrow \beta,$$

if and only if there exist strings δ_1, δ_2, and γ, and a non-terminal $N \in V_N$, such that

$$\alpha = \delta_1 N \delta_2, \quad \beta = \delta_1 \gamma \delta_2, \quad (N, \gamma) \in P.$$

- **Derivable**: A string β is said to be *derivable* from a string α, written as

$$\alpha \overset{*}{\Rightarrow} \beta,$$

if and only if either $\alpha = \beta$, or there exists a string γ such that

$$\alpha \overset{*}{\Rightarrow} \gamma \quad \text{and} \quad \gamma \Rightarrow \beta.$$

This means that a string is derivable from another string if we can reach the second string from the first through zero or more production steps.

- **Sentential form**: A sentential form of a grammar G is defined as

$$\alpha \mid S \overset{*}{\Rightarrow} \alpha$$

which is any string that is derivable from the start symbol S of G. Note that may be the empty string.

- **Terminal production**: A terminal production of a grammar G is defined as a sentential form that does not contain non-terminals:

$$\alpha \mid S \overset{*}{\Rightarrow} \alpha \wedge \alpha \in V_T^*.$$

- **Language generated by a grammar** G: The language $\mathcal{L}$ generated by a grammar G is defined as

$$\mathscr{L}(G) = \{\alpha \mid S \overset{*}{\Rightarrow} \alpha \wedge \alpha \in V_T^*\}.$$

So, the set of all terminal productions of G.

If G is a grammar for a programming language, then $\mathscr{L}(G)$ is the set of all programs in that language that are correct in a context-free sense.

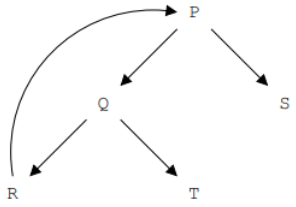
- **Sentences**: These terminal productions are called **sentences** in the language $\mathscr{L}(G)$
- **Intro to closure algorithms**: Quite a number of algorithms in compiler construction start off by collecting some basic information items and then apply a set of rules to extend the information and/or draw conclusions from them. These "information-improving" algorithms share a common structure which does not show up well when the algorithms are treated in isolation; this makes them look more different than they really are. We will therefore treat here a simple representative of this class of algorithms, the construction of the calling graph of a program, and refer back to it from the following chapters
- **Calling graph**: The calling graph of a program is a directed graph which has a node for each routine (procedure or function) in the program and an arrow from node A to node B if routine A calls routine B directly or indirectly. Such a graph is useful to find out, for example, which routines are recursive and which routines can be expanded in-line inside other routines

  The initial calling graph is, however, of little immediate use since we are mainly interested in which routine calls which other routine directly or indirectly. For example, recursion may involve call chains from A to B to C back to A. To find these additional information items, we apply the following rule to the graph: If there is an arrow from node A to node B and one from B to C, make sure there is an arrow from A to C.

  Consider the program

```
0  void P(void) { ... Q(); ... S (); ... }
1  void Q(void) { ... R(); ... T (); ... }
2  void R(void) { ... P (); }
3  void T(void) { ... }
4  void S(void) { ... }
```

  The calling graph is then

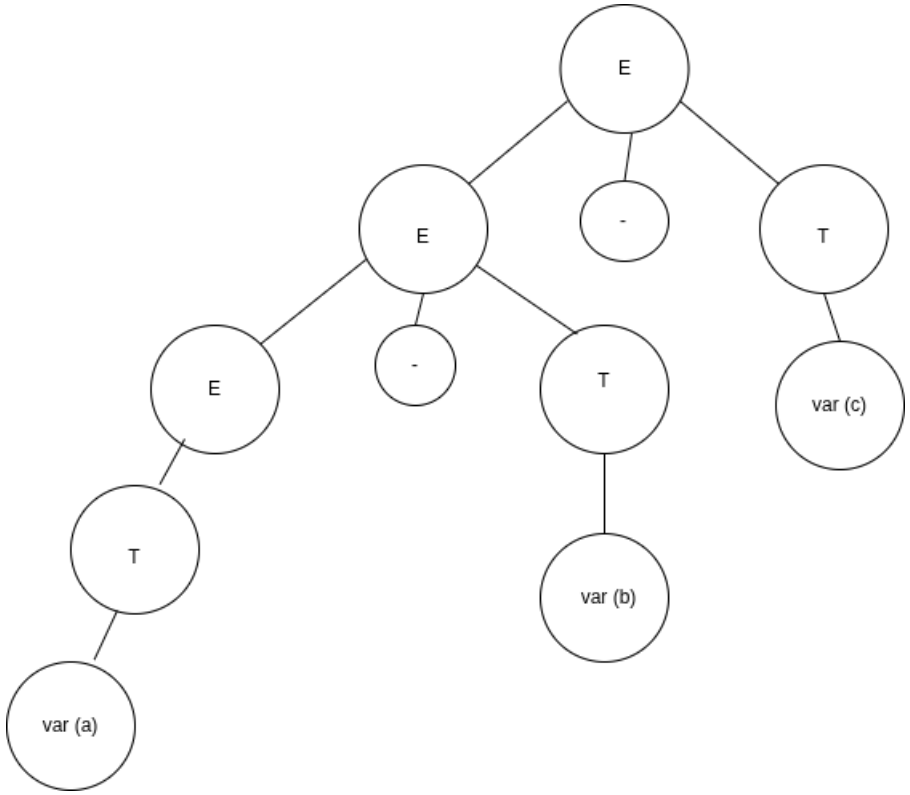


- **Transitive closure**: If we consider this rule as an algorithm (which it is not yet), this set-up computes the transitive closure of the relation "calls directly or indirectly". The transitivity axiom of the relation can be written as:

$$A \subseteq B \wedge B \subseteq C \rightarrow A \subseteq C,$$

  where the operator $\subseteq$ should be read as "calls directly or indirectly". Now, A is recursive is equivalent to $A \subseteq A$.

Adding this rule to the figure above, we get



We see that the recursion of the routines P, Q, and R has been brought into the open.

- **Components of a closure algorithm**:
  - **Data definitions**: definitions and semantics of the information items; these derive from the nature of the problem.
  - **Initializations**: one or more rules for the initialization of the information items; these convert information from the specific problem into information items.
  - **Inference rules**: one or more rules of the form: "If information items I1,I2,... are present then information item J must also be present". These rules may again refer to specific information from the problem at hand.

  The rules are called inference rules because they tell us to infer the presence of information item J from the presence of information items $I_1, I_2, ...$ When all inferences have been drawn and all inferred information items have been added, we have obtained the closure of the initial item set. If we have specified our closure algorithm correctly, the final set contains the answers we are looking for. For example, if there is an arrow from node A to node A, routine A is recursive, and otherwise it is not. Depending on circumstances, we can also check for special, exceptional, or erroneous situations

- **Recursion detection as a closure algorithm**:
  - **Data definitions:**
    1. G, a directed graph with one node for each routine. The information items are arrows in G.

2. An arrow from a node A to a node B means that routine A calls routine B directly or indirectly.

- **Initializations:** If the body of a routine A contains a call to routine B, an arrow from A to B must be present.
- **Inference rules:**: If there is an arrow from node A to node B and one from B to C, an arrow from A to C must be present.

Two things must be noted about this format. The first is that it does specify which information items must be present but it does not specify which information items must not be present; nothing in the above prevents us from adding arbitrary information items. To remedy this, we add the requirement that we do not want any information items that are not required by any of the rules: we want the smallest set of information items that fulfills the rules in the closure algorithm. This constellation is called the **least fixed point** of the closure algorithm.

The second is that the closure algorithm as introduced above is not really an algorithm in that it does not specify when and how to apply the inference rules and when to stop; it is rather a declarative,

- **Transitive closure algorithms**: General closure algorithms may have inference rules of the form "If information items $I_1, I2_2, ...$ are present then information item J must also be present", as explained above. If the inference rules are restricted to the form "If information items (A, B) and (B, C) are present then information item (A, C) must also be present", the algorithm is called a transitive closure algorithm

Lexical vs Syntactic analysis

- **Lexical analysis**: First phase of compilation, its purpose is to convert the raw input text (source code) into a sequence of tokens.
  - Reads characters from the source program
  - Groups characters into tokens
  - Removes whitespace and comments
  - Classifies lexemes into categories such as:
    * keywords
    * identifiers
    * literals
    * operators

  Consider

  ```
  x = 10 + y;
  ```

  The tokens are

  $$\text{IDENTIFIER ASSIGN NUMBER PLUS IDENTIFIER SEMICOLON}$$

- **Syntactic analysis**: Syntactic analysis (parsing) is the second phase of compilation. Its purpose is to determine whether the token sequence follows the grammar of the language.
  - Takes tokens from the lexer
  - Checks grammatical structure
  - Builds a parse tree or syntax tree
  - Detects syntax errors

  Checks whether

  ```
  x = 10 + y
  ```

  matches the grammar rule

  $$\text{assignment} \rightarrow \text{identifier} = \text{expression};.$$

  The component that performs syntactic analysis is called the parser.

- **Interpreters and compilers**
  - **Interpreters**: Lexical analysis → parsing → execute
  - **Compilers**: Lexical analysis → parsing → code generation → executable

- **Chomsky hierarchy**: The Chomsky Hierarchy is a classification of formal grammars based on their generative power—that is, the types of languages they can describe and the computational models required to recognize them.

It consists of four levels, ordered from most restrictive to most powerful.

- **Type 3 - Regular grammars**: Of the form

$$A \to aB \quad \text{or} \quad A \to a.$$

  Recognized by finite automata (DFA / NFA). Examples are regular languages, programming language tokens, and identifiers, numbers, keywords

- **Type 2 - CFGs**: Of the form

$$A \to \alpha,$$

  where

  * A is a single non-terminal
  * α is any string of terminals and non-terminals
  * Allows recursion
  * Can represent nested structures
  * Most programming language syntax is CFG-based

  Recognized by push down automata (PDA). Examples are

  * Arithmetic expressions
  * Balanced parentheses
  * Programming language syntax

- **Type 1 - CSGs**: Of the form

$$\alpha A \beta \to \alpha \gamma \beta$$

  with

$$|\gamma| \geqslant 1.$$

  Characteristics are

  * Rules depend on surrounding context
  * Length of strings never decreases
  * More powerful than CFGs

  Recognized by Linear bounded automata (LBA)

- **Type 0 - Unrestricted grammars**: Of the form

$$\alpha \to \beta,$$

  where α contains at least one non-terminal. Characteristics are

  * No restrictions on production rules
  * Most powerful grammar type
  * Can generate all computable languages

  Recognized by turing machines. Example is an REL.

- **Regular grammars / regular languages**: A regular grammar is the most restrictive class in the Chomsky hierarchy. It generates exactly the regular languages, which are the languages recognized by finite automata.

A grammar is regular if all of its productions are of one of the following forms:

- **Right regular**:

$$A \to aB \quad \text{or} A \to a.$$

- **Left regular**:

$$A \to Ba \quad \text{or} A \to a.$$

A grammar must be either entirely right-regular or entirely left-regular — never mixed.

Consider a mixed grammar

$$S \to \varepsilon \mid aA$$
$$A \to Sb.$$

This would allow

- Non-terminals on both sides
- Multiple derivation directions
- Power beyond regular languages

Such grammars can generate non-regular languages, which breaks the definition.

Notice that the above grammar could generate the language $\mathscr{L} = \{a^n b^n \ : \ n \geqslant 0\}$, which is famously nonregular. Observe that

$$S \to \varepsilon \mid aA$$
$$A \to bS$$

yields $\mathscr{L} = \{(ab)^n \ : \ n \geqslant 0\}$, which is regular. Notice that all productions are right-linear

- **Regular languages and FAs in lexical analysis**: Lexical analysis is the first phase of compilation. Its purpose is to convert a stream of characters into a stream of tokens.

  This process is based almost entirely on finite automata. Lexical structure of programming languages is:

  - Regular
  - Pattern-based
  - Non-nested
  - Locally decidable

  All of these can be described by regular expressions, which implies regular languages, which implies FAs.

- **Token specification**: Each token class is defined using an regular expression. For example,

$$\text{identifier} \to \text{letter}(\text{letter} \mid \text{digit})^*$$
$$\text{number} \to \text{digit}^+$$
$$\text{whitespace} \to (\text{space} \mid \text{tab} \mid \text{newline})^+.$$

  Each regular expression is converted into an NFA using standard constructions like Thompson's construction with ϵ-transitions allowed.

Note: Note efficient to execute directly, we can instead convert the NFA to a DFA, which is possible.

- **How the DFA is used (maximal munch / longest match rule)**:
  1. Start at the initial state
  2. Read input character by character
  3. Follow transitions
  4. Track the last accepting state
  5. When no transition is possible:
     - Backtrack to last accepting state
     - Emit the corresponding token
     - Restart from next input position

Note: You use a finite automaton to extract the tokens. The FA operations on raw characters.

$$\text{Char stream} \rightarrow \text{FA} \rightarrow \text{tokens}.$$

So the FA's job is to recognize token boundaries, not to consume tokens. The outputs of the FA is the tokens.

- **Lexical analyzer example**: We will build a lexer for the following token types

$$\begin{aligned} \text{ID} &\rightarrow [a-zA-Z][a-zA-Z0-9]^* \\ \text{NUM} &\rightarrow [0-9]^+ \\ \text{PLUS} &\rightarrow + \\ \text{WS} &\rightarrow \text{␣}^+. \end{aligned}$$

We will use the following transition table

Current	Letter	Digit	+	Whitespace
q0	q1	q2	q3	q0
q1	q1	q1	—	accept
q2	—	q2	—	accept
q3	—	—	—	accept

Consider the stream

```
sum1 + 42
```

Then, reading the stream gives

Input Read	State	Action
s	q1	continue
u	q1	continue
m	q1	continue
1	q1	continue
space	—	emit ID(sum1)
+	q3	emit PLUS
space	—	ignore
4	q2	continue
2	q2	continue
EOF	—	emit NUM(42)

So, the output tokens are

```
<ID, "sum1">
<PLUS, "+">
<NUM, "42">
```

- **Invalid tokens**: A token is invalid if:
  - The input character sequence does not match any token pattern
  - The DFA reaches a state with no valid transition
  - No accepting state was reached before failure

  In this case, a lexical error is reported.
- **Where do the tokens go**: The output of the lexical analyzer goes directly to the parser. The lexer produces a stream of tokens, and these tokens are consumed one-by-one by the parser. The lexer does not store the full list of tokens permanently — it typically supplies them on demand to the parser.
- **String to int conversion**

```cpp
int string_to_int(const string& s) {
    int i = 0;
    bool negative = false;

    if (s[0] == '-') negative = true;

    for (const auto& c : s)  {
        if (isdigit(c)) { i = (i * 10) + (c - '0'); }
        else return -1;
    }

    return negative ? -i : i;
}
```

- **Hex string to int conversion**:

```cpp
 int hex_stoi(const string& s) {
    int i = 0;
    for (const auto& c : s) {
        if (isdigit(c)) {
            i = i * 16 + (c - '0');
        } else if (c >= 97 && c <= 102) {
            i = i * 16 + (c - 'a') + 10;
        } else if (c >= 65 && c <= 70) {
            i = i * 16 + ((c - 'A') + 10);
        } else return -1;
    }

    return i;
}
```

- **UTF-8**
- **Recursion in CFGs**: It is important to note that when constructing CFGs, we should use only left-recursion or only right-recursion. If a grammar is is both left and right recursive,
  - Ambiguous parse trees
  - Infinite recursion in top-down parsers
  - No fixed associativity
  - Impossible to assign precedence cleanly
  - AST construction becomes ill-defined
- **Associativity in grammars**: Consider the grammar

$$E \to E + T \mid E - T \mid T$$
$$T \to \text{var} \mid \text{int}.$$

  For an expression like $a - b - c$, there are two interpretations,
  - **Left-associative**

$$(a - b) - c.$$

- **Right-associative**

$$a - (b - c).$$

Note that most arithmetic operators are left-associative. If we notice the rule $E \to E - T$, the recursive call to E is on the left-hand side of the production. This is called left recursion. Let's derive

$$a - b - c.$$

The derivation is

$$E \implies (E - T).$$

Then, expand the left E again,

$$E - T \implies ((E - T) - T) \implies ((T - T) - T) \implies ((a - b) - c).$$

This associativity is forced by the grammar. In order to derive

$$(a - (b - c))$$

we would need the rule

$$E \to T - E.$$

- **What's the point of a parse tree**: A parse tree gives you the complete syntactic structure of an input string as dictated by a grammar. More precisely, it shows how the grammar derives the string, step by step.

  A parse tree is a concrete representation of a derivation in a context-free grammar. It tells you:

  - Which grammar rules were applied
  - In what order they were applied
  - How the input string is grouped syntactically
  - How associativity and precedence are enforced

  Every internal node corresponds to a nonterminal, every leaf corresponds to a terminal symbol.

  A parse tree proves that a string:

  - belongs to the language
  - is generated by the grammar
  - If a parse tree exists → the string is syntactically valid.

- **Parsing expressions with grammars**: In compiler theory, a grammar serves three closely related goals:

  - Define the legal syntax of expressions.
  - Guide parsing to produce a parse tree (concrete syntax tree).
  - Enable construction of an AST, which is used for semantic analysis and code generation.

  The grammar must encode:

– Operator precedence
– Operator associativity
– Grouping rules (parentheses)

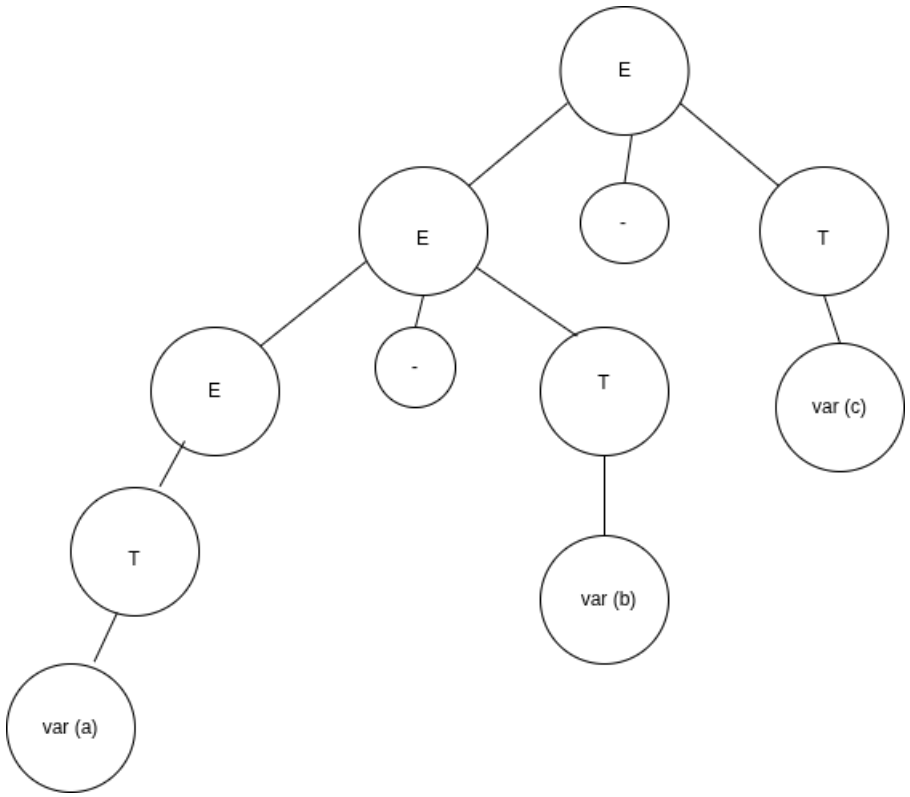
Consider the simple grammar

$$E \to E + T \mid E - T \mid T$$
$$T \to \text{var} \mid \text{int}.$$

Consider the expression $a - b - c$, the derivation is

$$E \implies E - T \implies E - T - T \implies T - T - T$$
$$\implies \text{var (a)} - \text{ var (b)} - \text{ var (c)},$$

thus yielding $a - b - c$. The parse tree from this derivation is then



Note that in parse trees, the children of a node are implicitly grouped as if surrounded by parentheses, Even if no parentheses appear in the source code, the tree structure itself acts as parentheses. A parse tree represents how the grammar groups subexpressions.

– Each node = an operation
– Its children = operands
– Subtrees = grouped expressions

Note that in derivations, when we expand a non-terminal, that expansion is implicitly grouped by parenthesis, even if we do not write the parenthesis in the derivation. In fact, we shouldn't. Every application of a production rule implicitly creates a grouped subtree, the grouping exists in the tree, not in the derivation string.

Suppose that we try to extend this grammar for multiplication and division, so we might extend our grammar to

$$E \to E + T \mid E - T \mid E * T \mid E/T \mid T$$
$$T \to \text{var} \mid \text{int}.$$

The problem is that this grammar treats all operators as equal precedence. So,

$$a + b * c$$

can be derived either as $((a + b) * c)$ or $(a + (b * c))$. For the first one, the derivation is

$$E \implies (E * T) \implies ((E + T) * T) \implies ((T + T) * T) \implies ((a + b) * c).$$

The second one is derived from

$$E \implies (E + T) \implies ((E * T) + T) \implies ((T * T) + T) \implies ((b * c) + a),$$

which is the same as

$$(a + (b * c)).$$

Both are valid derivations, which yield different parse trees for the same expression. This implies an ambiguous grammar, which is unacceptable for a complier.

Compilers must:

- Produce exactly one meaning
- Generate deterministic code
- Respect mathematical precedence

Ambiguity causes:

- Undefined behavior
- Multiple possible ASTs
- Impossible semantic analysis

So we need a grammar that forces precedence structurally, not by rules outside the grammar.

The solution is to separate precedence levels into different nonterminals.

- **Structural layers**: In order to enforce precedence, we must build structural layers in our grammar, where higher precedence operators appear on lower levels. Consider
  - **E:** Expression, lowest precedence
  - **T:** Term, medium precedence
  - **F:** Factor, highest precedence

  Each layer corresponds to how tightly operators bind. Precedence is enforced by how deep an operator appears in the grammar. Lower in the grammar enforces
  - more deeply nested in the parse tree
  - evaluated earlier
  - higher precedence

The grammar

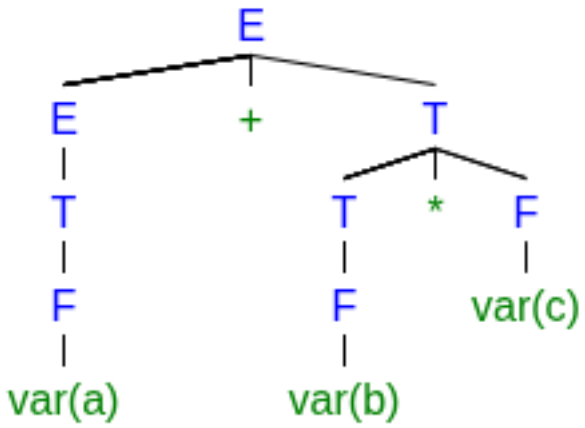
$$E \to E + T \mid E - T \mid T$$
$$T \to T * F \mid T/F \mid F$$
$$F \to \text{int} \mid \text{var}.$$

Provides correct operator precedence. Consider the expression $a+b*c$. The derivation is then

$$\begin{aligned} E &\implies E+T \implies T+T \implies T+T*F \implies T+T*\text{var (c)} \\ &\implies F+F+\text{var (c)} \implies \text{var (a)} \ +F+ \ \text{var (c)} \\ &\implies \text{var (a)} \ +\text{var (b)}+ \ \text{var (c)}. \end{aligned}$$

Recall that the parenthesis are not required in the derivation, as they are implicit and expressed by the parse tree. Derivations describe rule application order. Parse trees describe grouping.

The parse tree for the above derivation would be



With this knowledge its easy to see that if we switched the precedence of +,-,*,/ so that +,- appears on a deeper level than *,/, then $a+b$ would be grouped and evaluated before multiplying by c.

- **Explicit Parenthesis, custom order of operations**: We can create a new terminal (E), so our grammar would become

$$E \to E + T \mid E - T \mid T$$
$$T \to T * F \mid T/F \mid F$$
$$F \to (E) \mid \text{int} \mid \text{var}.$$

Note that parenthesis has highest precedence, so it must appear on the deepest level. Now, suppose we want the expression

$$(a + ((b - c) * d)).$$

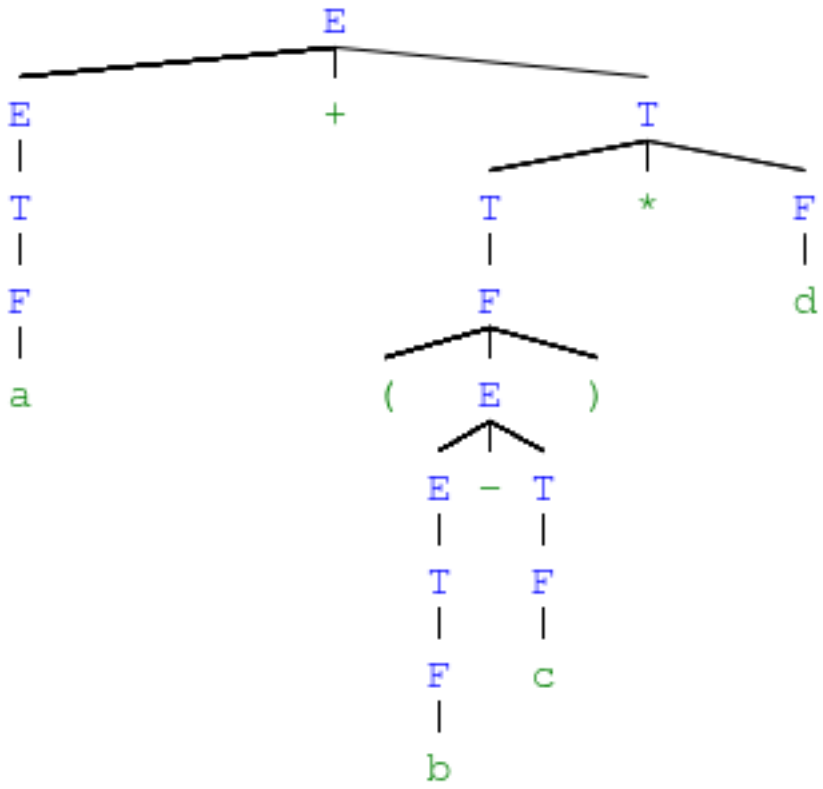
Note that most of these parenthesis are implicit i.e we do not need them, the only explicit parenthesis are

$$a + (b - c) * d.$$

However, even if parenthesis are usually implicit, but still added in the input string, we must use (E) to derive incorporate those parenthesis. For $a + (b - c) * d$ the derivation is

$$\begin{aligned} E &\implies E + T \implies E + T * F \implies E + F * F \implies E + (E) * F \\ &\implies \cdots \implies a + (E) * d \implies a + (E - T) * d \implies \cdots \implies a + (b - c) * d, \end{aligned}$$

which has parse tree



If the input string was instead

$$(a + ((b - c) * d)),$$

the derivation is

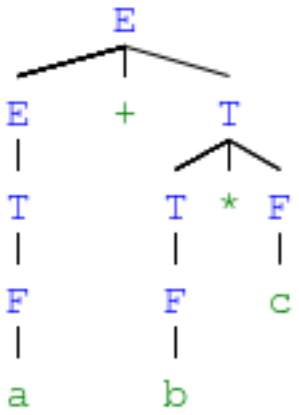
$$\begin{aligned} E &\implies T \implies F \implies (E) \implies (E + T) \implies (E + F) \implies (E + (E)) \\ &\implies (E + (T)) \implies (E + (T * F)) \implies (E + (F * F)) \implies (E + ((E) * F)) \\ &\implies (E + ((E - T) * F)) \implies \cdots \implies (a + ((b - c) * d)). \end{aligned}$$

- **AST's, parse tree to AST**: Turning a parse tree into an Abstract Syntax Tree (AST) is one of the most important conceptual steps in a compiler. The key idea is: A parse tree represents syntax. An AST represents meaning.

A parse tree includes

- Every nonterminal (E, T, F)
- Every production rule used
- Parentheses as grammar artifacts
- Structural nodes that carry no semantic meaning

The parse tree for $a + b * c$ is



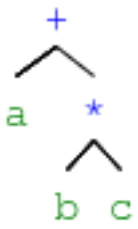
This tree

- Is correct
- Is verbose
- Contains nodes that exist only to enforce grammar rules

But a compiler does not need most of this information. An AST

- Removes grammar-specific nodes
- Keeps only semantic structure
- Represents computation directly
- Is independent of parsing strategy

For $a + b * c$, the AST is simply



If a node exists only to enforce grammar structure, remove it. If a node represents an operation or value, keep it.

To convert an parse tree to AST, we follow the following steps

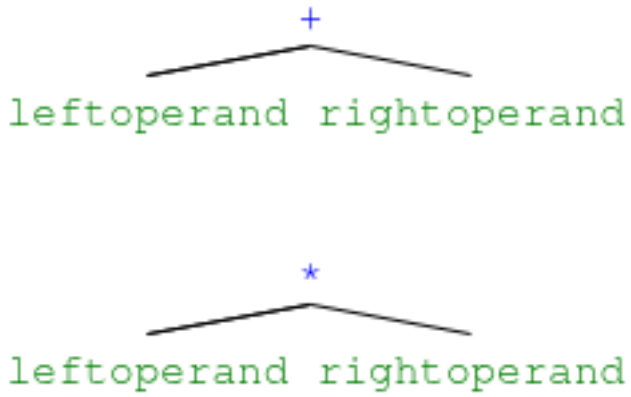
1. **Remove Non-semantic Nodes**: Nodes like E, T, F, parenthesis, and single child chains are removed.

   $$E \to T \to F \to a$$

   becomes simply a.
2. **Promote operators**: Nodes like

   $$E \to E + T,$$
   $$T \to T * F$$

   become



3. **Preserve hierarchy**: Because the grammar enforced precedence, the tree already has correct nesting, no additional rules are needed.

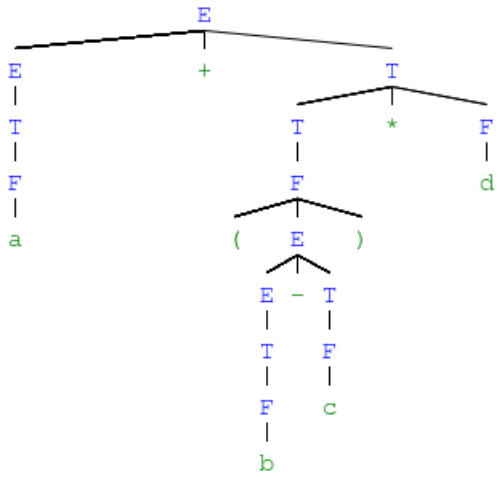
A parse tree answers "how was this derived?" An AST answers "what does this compute?"

In essence, to build an AST, remove grammar-only nodes from the parse tree and retain only operators and operands arranged according to the tree's structure.

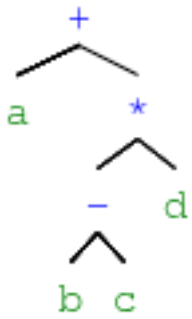
Algorithmically,

1. Visit parse tree node
2. If node is:
   - Operator → create AST node
   - Literal → create leaf
   - Grammar-only node → skip
3. Recursively process children
4. Return AST node upward

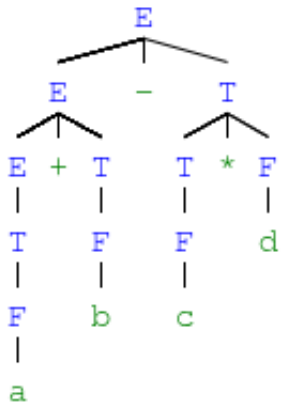
Notice that the parse tree



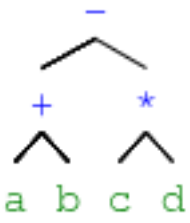
becomes the AST



If the expression $a + b - c * d$ had no explicit parenthesis, the parse tree would be



So, the AST would be



- **Evaluating ASTs**: A post-order traversal of an expression AST produces Reverse Polish Notation (RPN).

  This is not a coincidence; it is a direct consequence of how ASTs represent computation.

  In an AST:

  - Internal nodes = operators
  - Leaves = operands
  - Children = arguments to the operator

  Post-order traversal visits:

  - Left subtree
  - Right subtree
  - Node itself

  This is precisely

  operand operand operator

  which is reverse polish notation.

- **Negation and exponentiation**: To introduce unary negation and exponentiation correctly, we must extend the grammar without breaking precedence or associativity. This requires adding one new level and being careful about recursion direction. Negation binds tighter than multiplication or division, but looser than exponentiation. Exponentiation binds looser than parenthesis. So, the grammar becomes

  $$E \to E + T \mid E - T \mid T$$
  $$T \to T * U \mid T/U \mid U$$
  $$U \to -U \mid P$$
  $$P \to F \wedge P \mid F$$
  $$F \to (E) \mid \text{int} \mid \text{var}.$$

  Note that exponentiation is right-associative, so right recursion works here.

  $$a \wedge b \wedge c = a \wedge (b \wedge c).$$

Intel x86-64 architecture and code generation

- **General purpose registers**: General-purpose registers store integers, pointers, loop counters, and intermediate results.

  The 32-bit registers are

  - EAX – Accumulator (frequently used for arithmetic and return values)
  - EBX – Base register
  - ECX – Counter (loops, shifts)
  - EDX – Data register (multiplication/division)
  - ESI – Source index (string operations)
  - EDI – Destination index (string operations)
  - EBP – Base pointer (stack frame)
  - ESP – Stack pointer

  The 64-bit registers are

  - RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP
  - R8–R15 (additional registers introduced in 64-bit mode)

  Each 64-bit register has accessible sub-registers. For example,

  $$\text{AX} \rightarrow \text{EAX} \rightarrow \text{AX} \rightarrow \text{AH/AL}.$$

  This backward compatibility is fundamental to x86 design.

- **3-bit register specifier**: Many x86 instructions include a 3-bit register specifier, which implicitly numbers the registers:

Encoding	Register
000	EAX
001	ECX
010	EDX
011	EBX
100	ESP
101	EBP
110	ESI
111	EDI

  Note: Exists only at the binary encoding level

- **Instruction pointer (RIP)**:
  - **EIP (32-bit) / RIP (64-bit)**: Holds the address of the next instruction to execute.

- **The return instruction (C3)**: The ret (return) instruction transfers control back to the calling function by restoring the instruction pointer from the stack. It is the logical inverse of the call instruction.
- **Dynamic code execution basic example (CPP) (JIT execution)**: We will
  - Allocate memory at runtime
  - Write machine code (Intel x86/x86-64 instructions) into that memory
  - Mark the memory as executable
  - Cast the address to a function pointer
  - Call it

  This is exactly what a JIT compiler does at a minimal level. Consider

```
unsigned char prog[50000];
prog[0] = 0xC3; // ret instruction

int value;
// Cast to a function pointer, then call, store returned
 ↪  value from call in value
value = ((int(*)(void))prog)();
```

  If we run this code, we will get a seg fault. This is because our memory block is only read/write, we need a way to mark the memory as executable. We can do this by using sys/mman.h to allocate memory through the operating system, and errno to check for errors.

- **sys/mman.h**: <sys/mman.h> declares the memory-mapping API. It allows a process to request, control, and manage virtual memory regions directly from the operating system.

  The function **mmap** maps a region of virtual memory.

```
void* mmap(
    void* addr,
    size_t length,
    int prot,
    int flags,
    int fd,
    off_t offset
);
```

  fd is an open file descriptor referring to a file or device, it declares "this mapping is backed by the contents of this file." It must be opened with permissions compatible with prot. The kernel uses it as the source of bytes for the mapping. offset specifies where in the file the mapping starts, in bytes.

  Returns a pointer to the mapped region. On failure, it returns MAP_FAILED.

  munmap Unmaps a previously mapped region.

```
int munmap(void* addr, size_t length);
```

After munmap, any access to the region is invalid.

mprotect Changes access permissions on an existing mapping.

```
int mprotect(void* addr, size_t len, int prot);
```

Commonly used to transition memory:

- **RW:** Write code
- **RX:** Execute code

This is critical for enforcing $W \oplus X$ (write xor execute) security.

The protection flags (prot) are

- **PROT_READ**: Readable
- **PROT_WRITE**: Writable
- **PROT_EXEC**: Executable
- **PROT_NONE**: No access

These specify what the CPU is allowed to do with the memory

The mapping flags (flags) are

- **MAP_PRIVATE**: Copy-on-write
- **MAP_SHARED**: Shared between processes
- **MAP_ANONYMOUS**: Not backed by a file
- **MAP_FIXED**: Place mapping at a fixed address (dangerous)

- **errno**: errno is a thread-local integer that reports why a system call failed. It is declared in

```
<errno.h>
```

System calls signal failure via return values, they set errno to a symbolic error code. The value persists until overwritten

```
void* p = mmap(...);
if (p == MAP_FAILED) {
    perror("mmap");
}
```

Only meaningful immediately after failure, not reset automatically. The common errno values with mmap are

- **ENOMEM**: Insufficient memory or address space
- **EACCES**: Permission denied (e.g., PROT_EXEC disallowed)
- **EINVAL**: Invalid flags, alignment, or parameters
- **EBADF**: Invalid file descriptor
- **ENODEV**: Unsupported mapping type

To convert errno to text, use perror or strerror

```
perror(const char*)
strerror(errno)
```

perror prints a textual description of the error code currently stored in the system variable errno to stderr. It's parameter is a pointer to a null-terminated string with explanatory message

- **DCE with mmap and errno**:

```
const int prog_size = 50000;
unsigned char* prog;

   prog = mmap(0, prog_size, PROT_EXEC  | PROT_READ  |
↪  PROT_WRITE, MAP_PRIVATE  | MAP_ANONYMOUS, -1, 0);

if (errno) {
    perror("mmap");
    return;
}
```

Having fd=-1 means "This mapping is not backed by any file." This is only valid when used together with MAP_ANONYMOUS. Without it, fd=-1 is an error.

Having addr=0 means that the memory placement is decided by the operating system.

Now, prog points to the start of our block of memory that is readable, writable, and executable. Remember, to free the memory we must call `munmap`

```
munmap(prog, prog_size);
```

Now, with

```
value = ((int(*)(void))prog)();
```

We can output the value of value, which will give us the current value in the accumulator RAX. The return value of a function is defined to be placed in the accumulator register

- prog points to executable memory
- (int(*)(void))prog reinterprets that address as a function
- () performs a call instruction
- The CPU jumps to prog
- Your machine code executes
- A ret instruction returns control to the caller
- The caller reads the return value from the accumulator
- That value is assigned to value

Remember that we have

```
prog[0] = 0xC3
```

I.e the ret instruction, which is the only instruction currently in our program.

```
cout << "Memory at: " << (int*) prog << '\n';
```

Note that if we left our program pointer as char*, C++ uses the overload for outputing C-style strings. Since we just want the address of the first byte of our program, we must cast it to a different pointer, in this case a pointer to an integer. A better way to go about doing this is

```
cout << "Memory at: " << static_cast<void*>(prog) << '\n';
```

- **The ModR/M byte**: The ModR/M byte is a mandatory operand-encoding byte used by many x86 instructions. It specifies which registers are used and whether an operand refers to a register or to memory, including the basic addressing mode.

  The ModR/M byte is one byte (8 bits), divided into three fields

  - **r /m**: Bits 0-2
  - **reg**: Bits 3-5
  - **mod**: Bits 6-7

  mod determines how the r/m field is interpreted:

  - **00**: Memory operand, no displacement (except special cases)
  - **01**: Memory operand + 8-bit displacement
  - **10**: Memory operand + 32-bit displacement (16-bit in 16-bit mode)
  - **11**: Register operand (no memory access)

  The reg field usually selects a register operand.

  - **000**: EAX
  - **001**: ECX
  - **010**: EDX
  - **011**: EBX
  - **100**: ESP
  - **101**: EBP
  - **110**: ESI
  - **111**: EDI

  In /r encodings, reg is a real operand. In /0–/7 encodings, reg is an opcode extension, not a register

  The r/m field selects:

  - A register if mod = 11
  - A memory addressing mode if mod $\neq$ 11

  In the second case, r/m selects a base addressing form. In 32/64-bit mode:

  - 000 [EAX]
  - 001 [ECX]
  - 010 [EDX]
  - 011 [EBX]
  - 100 SIB byte follows
  - 101 [disp32] if mod=00, otherwise [EBP]+disp
  - 110 [ESI]
  - 111 [EDI]

  If r/m = 100 and mod $\neq$ 11, a SIB (Scale–Index–Base) byte follows. This allows

  $$[\text{base} + \text{index} \cdot \text{scale} + \text{displacement}].$$

- **Word sizes in Intel architecture**:
  - **Byte**: 8 bits
  - **Word**: 16 bits
  - **Doubleword (dword)**: 32 bits
  - **Quadword (qword)**: 64 bits
  - **Double quadword (dqword / xmmword)**: 128 bits
  - **YMM word**: 256 bits
  - **ZMM word**: 512 bits
- **MOV instructions: Putting a value into the accumulator**: For this we use the instruction

```
MOV     dst,src
```

Many forms of the MOV instruction can be found in the Intel manual. There are different forms based on whether dst, src is memory or a register, and how many bits these fields are. For example, we could use

```
MOV    r/m8,r8
```

where r/m8,r8 means that the destination operand can be either an 8-bit memory location, or an 8-bit register, and the source must be an 8-bit register. Conversely,

```
MOV    r8,r/m8
```

is the opposite. In the first form, the opcode is 88 /r, where /r denotes that the encoding requires a modR/M byte.

Note that there are forms for 8, 16, 32, and 64 bit. There are also immediate byte forms, for example

```
MOV    rk,immk
```

where $k \in \{8, 16, 32, 64\}$. Consider the form

```
MOV r32,imm32
```

This form has opcode $B8+$ rd id, where the d stands for *double word* 32-bit. If our instruction is

```
MOV    EAX,4
```

where 4 is the immediate byte, then the opcode is

$$B804000000.$$

Note that EAX is 000, so B8 remains the same. Further note that our immediate byte is 32-bits, since we used the double word version, and those bytes are in little endian, so the bytes are reversed. In BE it would be

$$00000004.$$

Another note is that there are no memory to memory moves, we must go memory to register, then register to memory.

Let's add this MOV instruction to our code, we add to the program buffer

```
size_t p_offset = 0;

prog[p_offset++] = 0xb8; // MOV EAX,4
prog[p_offset++] = 0x04;
prog[p_offset++] = 0x00;
prog[p_offset++] = 0x00;
prog[p_offset++] = 0x00;
prog[p_offset++] = 0xc3; // RET to caller
```

Now, the value returned from the call to prog gives the value 4, as 4 is the value in the accumulator.

- **A note**: The 64-bit architecture is called IA-32e, since it is an extension of the 32-bit architecture, not a 64-bit rework.
- **Two's complement**: IA uses twos complement, so to put -7 into the accumulator, we use

  $$f9ffffff.$$

  as the immediate byte. Notice the LE.
- **Finite width arithmetic**: An n-digit base-b register can store exactly

  $$\{0, 1, \ldots, b^n - 1\},$$

  nothing else. Any computation that exceeds this range wraps. The hardware drops the carry out of the top digit. This behavior is precisely arithmetic modulo b^n.
- **Twos complement**: This system comes from the question... On an n-digit base b-machine, what value should represent $-x$ so that subtraction can be done by addition.

  So, with finite-width arithmetic, we want a value y such that

  $$x + y \equiv 0 \pmod{b^n}.$$

  So, y is the additive inverse of x modulo b^n. Notice that since $x + y \equiv 0 \pmod{b^n}$,

  $$x + y = kb^n$$

  for some integer k. However, since

  $$0 \leqslant x, y \leqslant b^n - 1$$

  implies that

  $$0 \leqslant x + y \leqslant 2(b^n - 1) = 2b^n - 2,$$

  the only multiples of b^n that $x + y$ can equal are 0 and b^n, so we can safely say that $k = 1$. Thus, our equation becomes

  $$x + y = b^n.$$

  From this, we can solve for y

  $$y = b^n - x.$$

  Therefore,

  $$x + (b^n - x) = b^n,$$

  since $x + y = b^n$, and $y = b^n - x$. Thus,

  $$x + (b^n - x) \equiv 0 \pmod{b^n}.$$

- **Why twos complement works**: Consider an n digit system (base b), and a number x in that system. If we subtract by b^n, we get

$$b^n - x.$$

Now, we can add zero to this subtraction

$$b^n - 1 - x + 1 = (b-1)(b^{n-1} + b^{n-2} + \cdots + b^0) - x + 1.$$

Notice that $b^n - 1$ becomes $(b-1)(b^{n-1} + b^{n-2} + \cdots + b^0)$. Further notice that $b^n - 1$ is the number with all digits equal to $b-1$. In base 2, this would be all bits set to 1.

This algebraic trick reveals that

$$b^n - 1 = (b-1)(b^{n-1} + \cdots + b^0).$$

From this, subtracting x gives the digitwise complement of x. In base two, this would mean flipping all the bits. Thus,

$$b^n - 1 - x + 1$$

is the digitwise complement plus one, where the plus one is a carry. This is the step that turns the digitwise complement into the true additive inverse in base b. We now understand that

$$\bar{x} = (b^n - 1) - x.$$

This is the digitwise (radix-$b-1$) complement. So, we know the complement and the additive inverse. Notice that all we need to do to turn the complement into the additive inverse is add one to the complement.

- **A cleaned up twos complement argument**: Consider a base-b, n-digit system, note that our context is finite width arithmetic, since this is how registers work. Thus, all arithmetic is taken modulo b^n, since

$$0 \leqslant x \leqslant b^n - 1$$

for all x in our system. Now consider

$$b^n - x$$

Now, add zero to this expression

$$b^n - 1 - x + 1.$$

Consider the quantity $b^n - 1$, we can factor out a $b-1$ to get

$$(b^n - 1) = (b-1)(b^{n-1} + b^{n-2} + \cdots + b^1 + b^0).$$

Observe that this quantity is a number in our system with all digits equal to $b-1$. For example, if $b = 2$, then this is precisely the expression that gives us an n-bit binary number with all bits set to one.

So, subtracting x from this quantity gives the radix $b-1$ (digitwise) complement. In base two this would mean flipping all bits in x. So,

$$b^n - 1 - x + 1$$

is the digitwise complement plus one. Our goal is to derive the additive inverse of x using this digitwise complement. The additive inverse of x in unbounded width arithmetic would be the number y such that

$$x + y = 0.$$

However, we are dealing with finite width arithmetic, so all expressions equal to zero must be congruent to zero when taken modulo b^n. So, we must instead consider

$$x + y \equiv 0 \pmod{b^n}.$$

Using modular arthritic, this means that

$$x + y = kb^n$$

for some $k \in \mathbb{Z}$. But, notice that since

$$0 \leqslant x, y \leqslant b^n - 1,$$

it must be that

$$0 \leqslant x + y \leqslant 2(b^n - 1) = 2b^n - 2.$$

Since the maximum in our system is b^n, the only possible multiple of b^n is b^n itself, so $k = 1$ (actually $k \in \{0, 1\}$, but $k = 0$ only if $x = y = 0$, otherwise $k = 1$). Thus,

$$x + y = b^n.$$

Now we see that the additive inverse y is

$$y = b^n - x.$$

Notice that the digitwise complement $(b^n - 1 - x)$ and the additive inverse $(b^n - x)$ look very similar. All we need to do to get from the digitwise complement to the additive inverse is add one to the complement, which is precisely what twos complement does.

The story of the sign bit (in base two) arises once you define the signed interpretation map

$$\phi(u) = \begin{cases} u, & 0 \leqslant u < 2^n - 1 \\ u - 2^n, & 2^{n-1} \leqslant u < 2^n \end{cases}.$$

Notice that

$$\phi(2^n - u) = (2^n - u) - 2^n = -u = -\phi(u).$$

Any value greater than or equal to 2^{n-1}, which corresponds to MSB=1 is negative, hence the sign bit.

In addition, consider performing twos complement twice on any number x,

$$2^n - (2^n - x) = x$$

so we get back x. This is why we don't have to reverse the process of twos complement to undo it.

- **ADD instructions: Adding the value from a register into the accumulator**: Suppose we want to add +3 into ECX, then add that value to the accumulator. We can use the ADD instruction version

```
ADD    r32, r/m32
```

with opcode 03 /r and the same immediate byte move instruction

```
MOV    ECX, 3
```

This has encoding

$$B903000000.$$

The ADD instruction is

```
ADD    EAX,ECX
```

The ModR/M byte for the ADD instruction is $C1$, since mod=11, reg=000 (ECX), and r/m = 001 (EAX). Thus, the encoding is

$$03C1.$$

- **Note on instruction length**: An x86 instruction can be anywhere from 1 to 15 bytes long. There is no fixed instruction width. The length depends on which optional components are present.
- **/0 - /7**: In Intel x86 architecture, the symbols /0 through /7 refer to opcode extensions encoded in the ModR/M byte of an instruction. They are not registers themselves; rather, they select which operation an instruction performs when multiple operations share the same primary opcode.

  Many x86 instructions reuse a single opcode and rely on the ModR/M byte to disambiguate the operation. When documentation writes /0, /1, …, /7, it means "The reg/opcode field of the ModR/M byte must contain this value." That 3-bit field can encode values from 0 to 7, hence /0–/7.

  Therefore, the 3-bit reg field has exactly one meaning per instruction:

  - Either it selects a register operand
  - Or it selects an opcode extension (/0–/7)
- **Substraction instructions**: Suppose we want to subtract ECX from EAX,

```
SUB    EAX, ECX
```

Suppose ECX has value 3, and EAX has value -7. The opcode for this SUB instruction

```
SUB r/m32, r32
```

Is 29 /r. The encoding is therefore $29C8$, since the ModR/M byte is C8 (11 001 000).

- **Multiplication instructions**: We have two primary instructions, MUL (unsigned multiplication), and IMUL (signed multiplication). Regarding IMUL, there are four one operand instructions, one for each register/memory size. Consider

```
IMUL    r/m32
```

It should be noted that the operand is multiplied against the value in the accumulator, and the DX/AX registers are combined to make one 64-bit register split into two 32-bit sections, 32-bits per half. The DX register is the high order 32-bits, and the AX register is the low order 32-bits. Since we will use the 32-bit version, EDX and EAX are our register pairs.

The opcode for this instruction is f7 /5, so we use 101 in the 3-bit reg field in the ModR/M byte.

If our instruction was

```
IMUL    ECX
```

,this byte would be 11 101 001, which is $E9$ in hex. Thus, the encoding is $F7E9$.

- **Division and converting instructions**: We have DIV for unsigned division, and IDIV for signed division. This operation divides the value in the accumulator by the value in the supplied operand, and again the result is split into the DX / AX pair. The quotient of the operation will be housed in AX, and the remainder in DX.

Before we can proceed with the division, we must extend the sign bit of the accumulator into the DX register. For this, we need CWD, CDQ, CQO instructions. CWD converts a word to a double word, and CDQ converts a double word to a quadword. In particular, the version of these instructions that extend the sign bit of AX into DX are given by opcode 99, and they take no operands. Thus,

```
CDQ
```

Extends the sign bit of EAX (32-bit doubleword) into EDX, thus converting to a quadword (64-bit). This operation must be done before division.

Note that the encoding for CDQ is 99, and the encoding for IDIV is f7 /7, where the instruction is

```
IDIV   r/m32
```

- **XOR instruction, Clearing registers**: The XOR instruction is

```
XOR   r /m32 r32
```

and variants. The opcode for this particular instruction is $31/r$. To clear a register, we therefore use the same register for both operands

- **R8-R15, the REX prefix**: The REX prefix is a one-byte instruction prefix introduced with x86-64 to extend the original x86 encoding so that it can address 64-bit operands and additional registers.

  The REX prefix is one byte, with the following fixed structure:

  ```
  0100WRXB
  ```

  The high nibble 0100 identifies it as a REX prefix. The low nibble contains four 1-bit flags.

Bit	Name	Purpose
REX.W	64-bit operand size	W
R	REX.R	Extends ModR/M reg field
X	REX.X	Extends SIB index field
B	REX.B	Extends ModR/M r/m or SIB base

  REX.W = 1 forces a 64-bit operand size. Requires REX.W Without it, the instruction would operate on 32-bit registers.

  REX.R, REX.X, and REX.B extend 3-bit register fields into 4-bit fields.

  - **REX.R**: Extends the reg field of the modR/m byte. Place the high order bit of the non r /m register mask into this bit. For example, for R8 = 1000, set R=1.
  - **REX.X**: Extends the index field in the SIB byte, only meaningful if a SIB byte is present, otherwise leave zero
  - **REX.B**: Extends the r /m field of the modr /m byte. Place the high order bit of the r / m register mask into this bit.

- **Recall: Fast exponentiation**: Uses the identity that

$$a^b = \begin{cases} (a^2)^{\frac{b}{2}} & b = 2k \\ a \cdot a^{b-1} & b = 2\ell + 1 \end{cases}.$$

```
e = 1
// b!=0 and b>=0 (nonnegative b) implies b>0
while (b>0) {
    if (b & 0x01) {
        e = e * a
    }
    a = a*a
    b>>=1 // Divide by two
}
```

- k **child trees to binary trees**: When creating abstract syntax trees, it is likely the case that a binary tree is too restrictive, we may need to have trees that can have k children, for k not restricted to 2. Thus, it is helpful to know the way in which we can convert such a tree into a binary tree.

  The idea is that for each parent node, we take its first (leftmost) child and make that its child in the binary tree. Then, for any other children (siblings of the first child), we attach those in a link to the first child. So, if A has children B, C, D, and B has children E, F, then the binary tree would have connections

  $$A \to B,$$
  $$B \to C, E$$
  $$C \to D,$$
  $$E \to F.$$

- **The POP and PUSH instructions**: The PUSH instruction pushes a word, doubleword, or quadword onto the stack. Decrements the stack pointer and then stores the source operand on the top of the stack.

```
PUSH    r32
```

  The POP instruction loads the value from the top of the stack to the location specified with the destination operand

```
POP     r32
```

- **The ECHG (exchange) instruction**: Exchanges register / memory with register
- **Machine execution model (MEM)**: There is a problem that arises when trying to generate machine code. We have many registers, but when generating code, which registers do we use? And if we have to go to memory if those registers are in use, which addresses do we go to? And how does this access of memory to store values work?

  Since this is outside the scope of our current progress in compiler theory, we will start with the use of a stack based model.

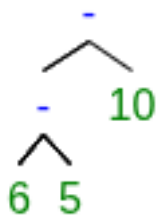
  In this model, we will have the top of the stack always be the current value in the accumulator.

First, consider a generic tree

where T_1 and T_2 are subtrees. When T_1 completes its duties, its computed value will be in the accumulator. So, we push this value onto the stack. Then, we can generate code for T_2 and its value will now be in the accumulator. From here, we need to perform the addition with these two values. So, we pop the top of the stack into ECX, then write the instruction

```
0  ADD    EAX,ECX
```

Consider the expression $6 - 5 - 10$, which generates the AST



The code will look something like

```
0  // Computes 6-5 and places result on the top of the stack
1  MOV   EAX,6
2  PUSH  EAX
3  MOV   EAX,5
4  POP   ECX
5  XCHG  EAX, ECX
6  SUB   EAX, ECX
7  PUSH  EAX
8
9  // Computes 6-5-10, 6-5 is on the top of the stack at this
   ↪  point
10 MOV   EAX,10
11 POP   ECX
12 XCHG  EAX,ECX
13 SUB   EAX,ECX
```

- **The TEST instruction**: The TEST instruction in IA-32e (the 64-bit extension of the x86 architecture) performs a bitwise logical AND between two operands without storing the result. Its sole purpose is to update the condition flags in RFLAGS. Given

```
TEST  op1, op2
```

The processor computes

```
temp := op1 AND op2
```

The value of temp is not written back to any operand. Instead, only flags are updated. After execution:

- **ZF (Zero Flag)**: Set if temp == 0, otherwise cleared.
- **SF (Sign Flag)**: Set according to the most significant bit of temp.
- **PF (Parity Flag)**: Set according to parity of the low byte of temp.
- **CF (Carry Flag)**: Cleared to 0
- **OF (Overflow Flag)**: Cleared to 0
- **AF (Auxiliary Flag)**: Undefined

This is identical to AND except the result is discarded.

- **Common use cases of TEST**:

  1. **Checking if a register is zero**:

     ```
     TEST   R,R

     // Jump on ZF zero
     jz
     ```

  2. **Testing specific bits**:

     ```
     test rax, 1
     ```

     This tests if the LSB is set (check ZF).

- **Code generation for the fast exponentiation algorithm**: We aim to compute the quantity $e = a^b$. For this, we will place

$$e \to r8, \quad a \to r9, \quad b \to r10.$$

Assume the stack has b on the bottom, and a at the top. The code is then

```
// Clear out r8
XOR   r8,r8

// Place |a| in r9
POP   r9

// Place |b| in r10
POP   r10

TEST r10,r10
```