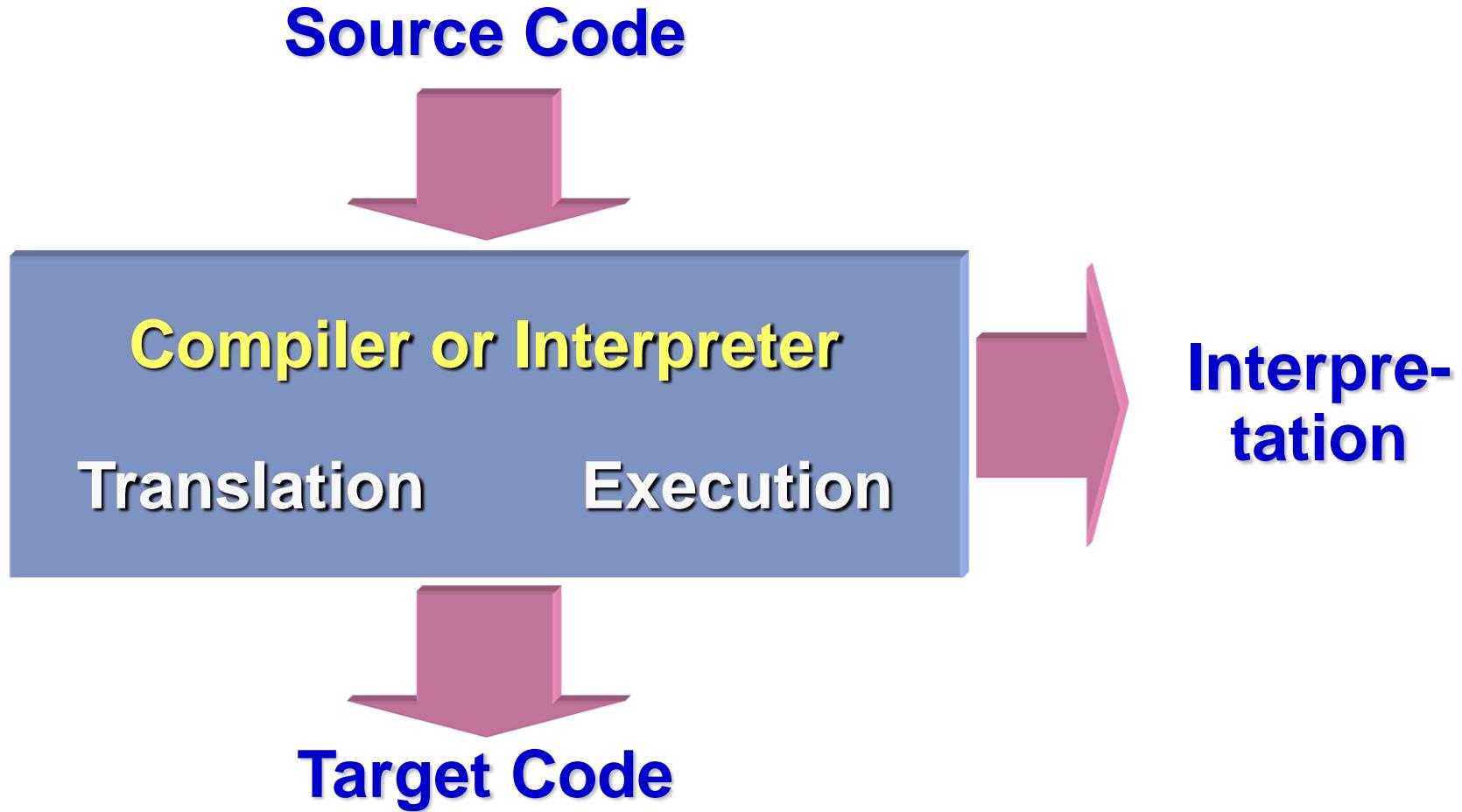

Programming Languages Advance

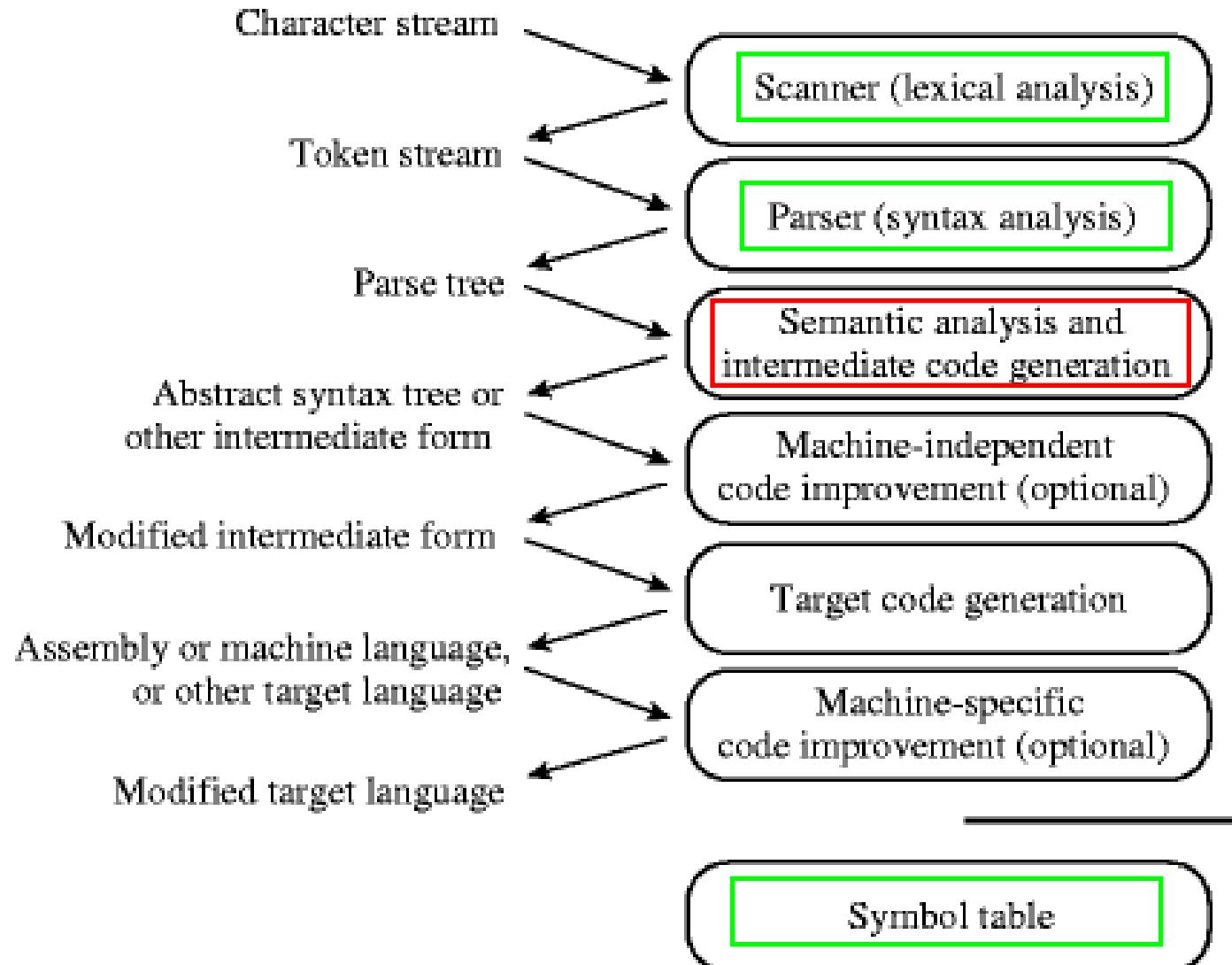
Semantic Analysis

Semantic Analysis

From Code Form To Program Meaning



Phases of Compilation



Specification of Programming Languages

- PLs require precise definitions (i.e. no ambiguity)
 - Language *form* (Syntax)
 - **Language *meaning* (Semantics)**
- Consequently, PLs are specified using formal notation:
 - Formal syntax
 - » Tokens
 - » Grammar
 - **Formal semantics**
 - » Attribute Grammars (static semantics)
 - » Dynamic Semantics

The Semantic Analyzer

- The principal job of the semantic analyzer is to enforce static semantic rules.
- In general, anything that requires the compiler to compare things that are separate by a long distance or to count things ends up being a matter of *semantics*.
- The semantic analyzer also commonly constructs a syntax tree (usually first), and much of the information it gathers is needed by the code generator.

Attribute Grammars

- Context-Free Grammars (CFGs) are used to specify the syntax of programming languages
 - E.g. arithmetic expressions
- How do we tie these rules to mathematical concepts?
- *Attribute grammars* are annotated CFGs in which *annotations* are used to establish meaning relationships among symbols
 - Annotations are also known as decorations

$E \rightarrow E + T$
$E \rightarrow E - T$
$E \rightarrow T$
$T \rightarrow T * F$
$T \rightarrow T / F$
$T \rightarrow F$
$F \rightarrow - F$
$F \rightarrow (E)$
$F \rightarrow \text{const}$

Attribute Grammars

Example

- Each grammar symbol has a set of *attributes*
 - E.g. the value of E_1 is the attribute $E_1.\text{val}$
- Each grammar rule has a set of rules over the symbol attributes
 - *Copy rules*
 - *Semantic Function rules*
 - » E.g. sum, quotient

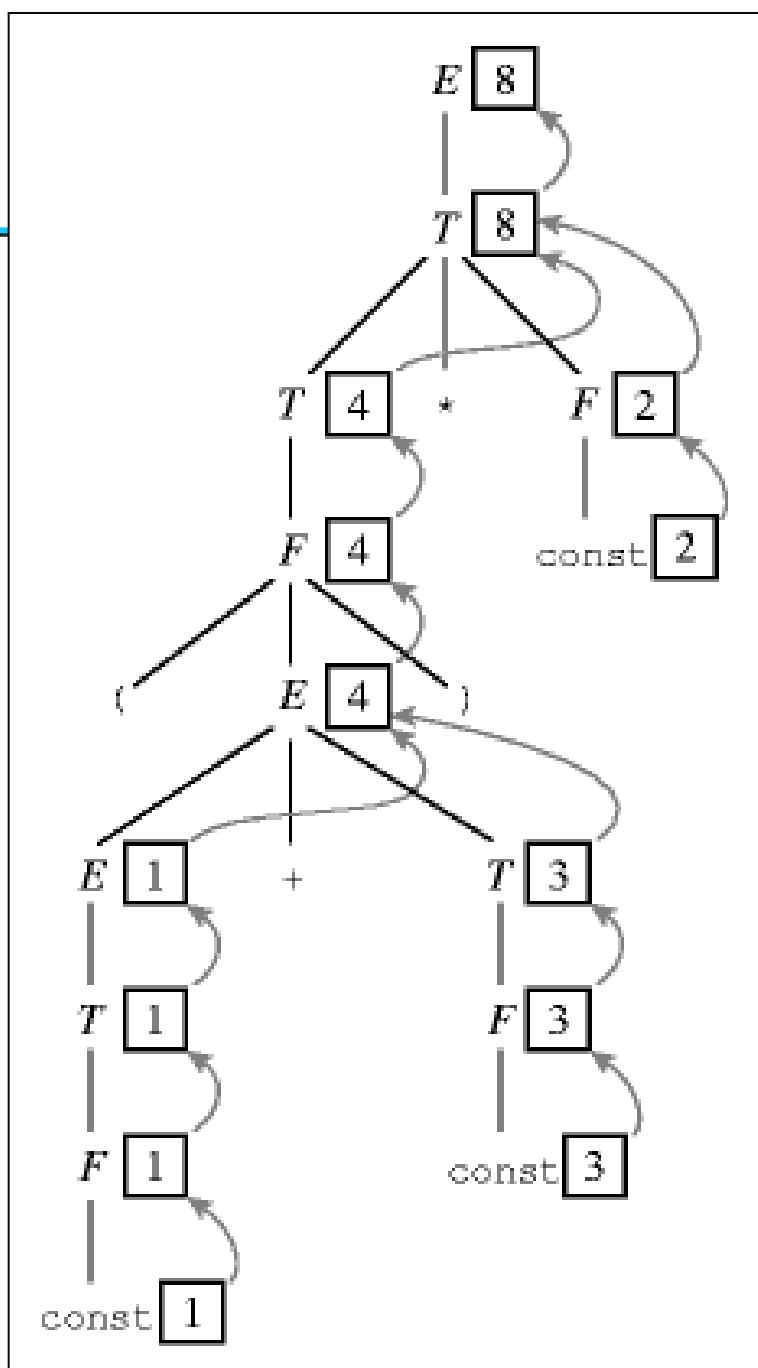
- 1: $E_1 \rightarrow E_2 + T$
▷ $E_1.\text{val} := \text{sum} (E_2.\text{val}, T.\text{val})$
- 2: $E_1 \rightarrow E_2 - T$
▷ $E_1.\text{val} := \text{difference} (E_2.\text{val}, T.\text{val})$
- 3: $E \rightarrow T$
▷ $E.\text{val} := T.\text{val}$
- 4: $T_1 \rightarrow T_2 * F$
▷ $T_1.\text{val} := \text{product} (T_2.\text{val}, F.\text{val})$
- 5: $T_1 \rightarrow T_2 / F$
▷ $T_1.\text{val} := \text{quotient} (T_2.\text{val}, F.\text{val})$
- 6: $T \rightarrow F$
▷ $T.\text{val} := F.\text{val}$
- 7: $F_1 \rightarrow - F_2$
▷ $F_1.\text{val} := \text{additive_inverse} (F_2.\text{val})$
- 8: $F \rightarrow (E)$
▷ $F.\text{val} := E.\text{val}$
- 9: $F \rightarrow \text{const}$
▷ $F.\text{val} := \text{const}.val$

Attribute Flow

- Context-free grammars are not tied to an specific parsing order
 - *E.g.* Recursive descent, LR parsing
- Attribute grammars are not tied to an specific evaluation order
 - This evaluation is known as the *annotation* or *decoration* of the parse tree

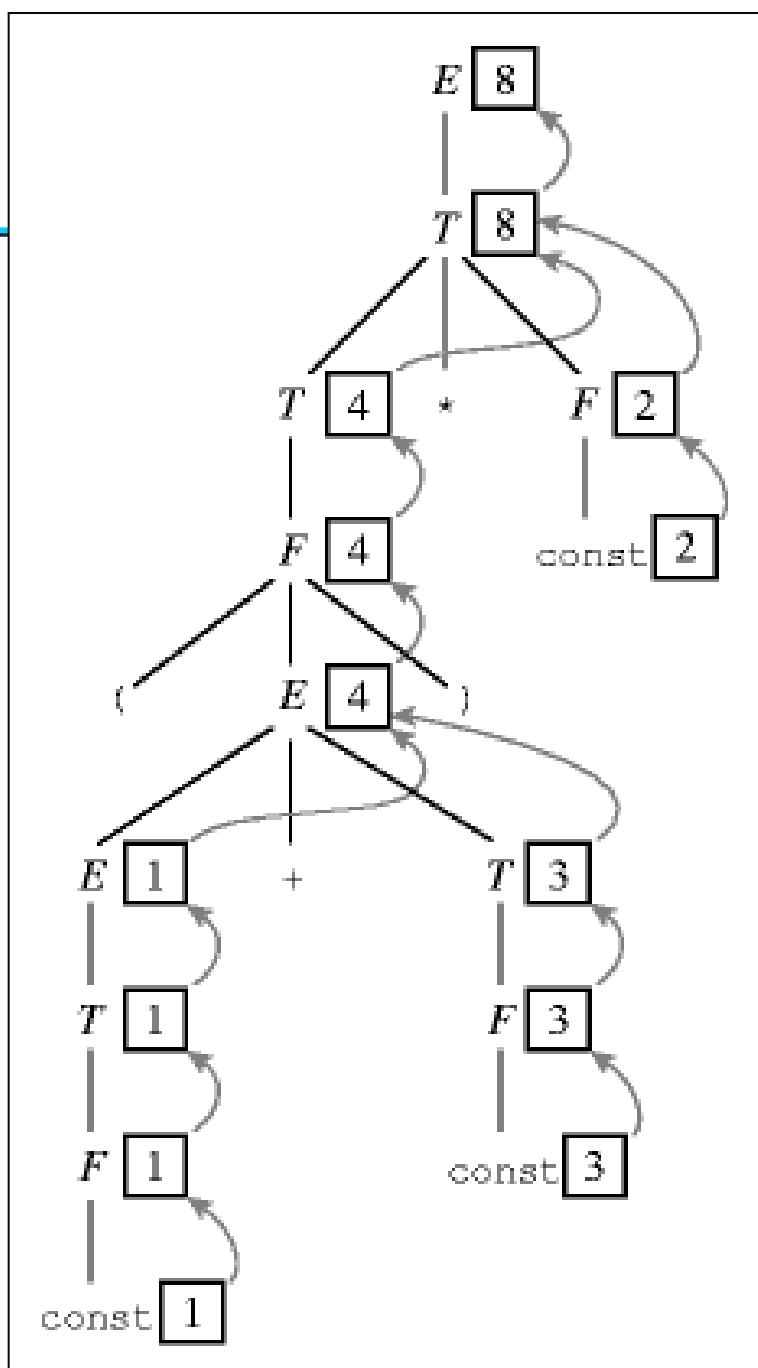
Attribute Flow Example

- The figure shows the result of annotating the parse tree for $(1+3) * 2$
- Each symbol has at most one attribute shown in the corresponding box
 - Numerical value in this example
 - Operator symbols have no value
- Arrows represent *attribute flow*



Attribute Flow Example

- 1: $E_1 \rightarrow E_2 + T$
▷ $E_1.\text{val} := \text{sum}(E_2.\text{val}, T.\text{val})$
- 2: $E_1 \rightarrow E_2 - T$
▷ $E_1.\text{val} := \text{difference}(E_2.\text{val}, T.\text{val})$
- 3: $E \rightarrow T$
▷ $E.\text{val} := T.\text{val}$
- 4: $T_1 \rightarrow T_2 * F$
▷ $T_1.\text{val} := \text{product}(T_2.\text{val}, F.\text{val})$
- 5: $T_1 \rightarrow T_2 / F$
▷ $T_1.\text{val} := \text{quotient}(T_2.\text{val}, F.\text{val})$
- 6: $T \rightarrow F$
▷ $T.\text{val} := F.\text{val}$
- 7: $F_1 \rightarrow -F_2$
▷ $F_1.\text{val} := \text{additive_inverse}(F_2.\text{val})$
- 8: $F \rightarrow (E)$
▷ $F.\text{val} := E.\text{val}$
- 9: $F \rightarrow \text{const}$
▷ $F.\text{val} := \text{const}.val$



Attribute Flow

Synthetic and Inherited Attributes

- In the previous example, semantic information is passed up the parse tree
 - We call this type of attributes are called *synthetic attributes*
 - Attribute grammar with synthetic attributes only are said to be *S-attributed*
- Semantic information can also be passed down the parse tree
 - Using *inherited attributes*
 - Attribute grammar with inherited attributes only are said to be *non-S-attributed*

Attribute Flow

Inherited Attributes

- *L-attributed* grammars, such as the one on the next slide, can still be evaluated in a single left-to-right pass over the input.
- Each synthetic attribute of a LHS symbol (by definition of *synthetic*) depends only on attributes of its RHS symbols.
- Each inherited attribute of a RHS symbol (by definition of *L-attributed*) depends only on inherited attributes of the LHS symbol or on synthetic or inherited attributes of symbols to its left in the RHS.
- Top-down grammars generally require non-S-attributed flows
 - The previous annotated grammar was an S-attributed LR(1)
 - L-attributed grammars are the most general class of attribute grammars that can be evaluated during an LL parse.

Syntax Tree

- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved.
- A *one-pass* compiler interleaves scanning, parsing, semantic analysis, and code generation in a single traversal of the input.
- A common approach interleaves construction of a syntax tree with parsing (eliminating the need to build an explicit parse tree), then follows with separate, sequential phases for semantic analysis and code generation.

Action Routines

- Automatic tools can construct a parser for a given context-free grammar
 - E.g. yacc
- Automatic tools can construct a semantic analyzer for an attribute grammar
 - An ad hoc techniques is to annotate the grammar with executable rules
 - These rules are known as *action routines*

Static and Dynamic Semantics

- Attribute grammars add basic semantic rules to the specification of a language
 - They specify *static semantics*
- But they are limited to the semantic form that can be checked at compile time
- Other semantic properties cannot be checked at compile time
 - They are described using *dynamic semantics*

Dynamic Semantics

- Use to formally specify the behavior of a programming language
 - Semantic-based error detection
 - Correctness proofs
- There is not a universally accepted notation
 - **Operational semantics**
 - » Executing statements that represent changes in the state of a real or simulated machine
 - **Axiomatic semantics**
 - » Using predicate calculus (pre and post-conditions)
 - **Denotational semantics**
 - » Using recursive function theory

Semantic Specification

- The most common way of *specifying* the semantics of a language is plain english
 - http://java.sun.com/docs/books/jls/first_edition/html/14.doc.html#24588
- There is a lack of formal rigor in the semantic specification of programming languages
 - Guess why

What is top-down parsing?

- Top-down parsing is a parsing-method where a sentence is parsed starting from the root of the parse tree (with the “*Start*” symbol), working recursively down to the leaves of the tree (with the terminals).
- In practice, top-down parsing algorithms are easier to understand than bottom-up algorithms.
- Not all grammars can be parsed top-down, but most context-free grammars can be parsed bottom-up.

Bottom-up Parsing

- Bottom-up parsers parse a programs from the leaves of a parse tree, collecting the pieces until the entire parse tree is built all the way to the root.
- Bottom-up parsers emulate pushdown automata:
 - requiring both a state machine (to keep track of what you are looking for in the grammar) and a stack (to keep track of what you have already read in the program).
 - making it fairly easy to automate the process of creating the parser
 - ensuring that all context-free grammars can be parsed by this method.

Bottom-up parsers as shift-reduce parsers

- Bottom-up parsers are frequently called shift-reduce parsers because of their two basic operations:
 - A shift involves moving pushing the current input token onto the stack and fetching the next input token.
 - A reduce involves popping all the variables that comprise the right-sentential form for a nonterminal and replacing them on the stack with the equivalent nonterminal that appears on the left-hand side of that production.
 - While shifting involve pushing and reducing involve popping, do not think of them as equivalent: a shift also involve advancing the input token stream and a reduce involves zero or more pops followed by a push.

Bottom-up Parsing as an Emulation of Pushdown Automata

- Most bottom-up parsers are table-driven, with the table encoding the necessary information about the grammar.
- The parser decides what action to perform based on the combination of current state and current input token.
- A state in the machine which the computer is emulating reflects both what the machine has already parsed and that which it is expect to see in the input token stream.
- Several parser generators have been created based on this theoretical machine, the best known of which is **YACC** (Yet Another Compiler Compiler), is available on many UNIX system and its public domain lookalike **Bison**.