Syntax and Semantics

Hui Chen

Computer Science
Virginia State University
Petersburg, Virginia

January 20, 2016

Outline

- Backus-Naur Form
  - derivations, parse trees, ambiguity, descriptions of operator precedence and associativity, and extended Backus-Naur Form.
- Attribute grammars
- Operational axiomatic and denotational semantics

Chomsky Hierarchy

- Also called Chomsky-Schützenberger Hierarchy (Noam Chomsky, 1956)

Class	Grammar	Language	Automaton
Type-0	Unrestricted	Recursively enumerable	Turing machine (TM)
Type-1	Context-sensitive	Context-sensitive	Linear-bounded automaton (LBA)
Type-2	Context-free	Context-free	Pushdown automaton (PDA)
Type-3	Regular	Regular	Deterministic finite automaton (DFA)

- A strictly nested sets of classes of formal grammars, i.e.,

$$\text{Type-0} \supset \text{Type-1} \supset \text{Type-2} \supset \text{Type-3}$$

- Context-free and regular grammars are of our primary concern

Context-Free Grammar (CFG)

- A CFG is a quadruple, $G = (V, T, P, S)$ where
  - V: the set of variables or non-terminals
  - T: the set of terminals
  - P: the set of productions of the form $A \to \gamma$ where A is a single variable, i.e., $A \in V$ and γ is string of terminals and variables, i.e., $\gamma \in (V \cup T)^*$
  - S: the start symbol and $S \in V$
- To describe the grammar of a programming language,
  - Terminals are lexemes or tokens

Example: A Simple Programming Language[1]

- Operators: $+$ and $*$ represent addition and multiplication, respectively
- Arguments are identifiers consisting *only* of letters a, b, and digits 0, 1
- An example statement in the language,

$$(a + b) * (a + b + 1)$$

[1] This is an example given in [Hopcroft et al., 2006]

CFG of the Simple Language

- The language can be specified using a CFG as,

$$G = (\{E, I\}, T, P, E)$$

where

- E and I are the two variables, and E is the start symbol
- T, the terminals are the set of symbols $\{+, *, (,), a, b, 0, 1\}$
- P is the productions, i.e.,

1. $E \to I$
2. $E \to E + E$
3. $E \to E * E$
4. $E \to (E)$
5. $I \to a$
6. $I \to a$
7. $I \to Ia$
8. $I \to Ib$
9. $I \to I0$
10. $I \to I1$

Backus-Naur Form (BNF)

- John Backus (1959) and Peter Naur (1960) developed to describe syntax of ALGOL 58 and 60
- BNF is equivalent to context-free grammars
- Widely used today for describing syntax of programming languages

Production Rules in BNF

- Nonterminals (or variables in CFG, called *abstractions*) are often enclosed in angle brackets
- A start symbol is a special element of the nonterminals of a grammar
- Grammar: a finite non-empty set of rules
- Examples of BNF rules:

$$<\text{ident\_list}> \to \text{identifier}$$
$$<\text{ident\_list}> \to \text{identifier}, <\text{ident\_list}>$$
$$<\text{if\_stmt}> \to \text{if} <\text{logic\_expr}> \ \text{then} <\text{stmt}>$$

More than one RHS

- An abstraction (or a nonterminal symbol) can have more than one right-hand sides
- Example: applying this rule, we can rewrite,

$$<\text{ident\_list}> \rightarrow \text{identifier}$$
$$<\text{ident\_list}> \rightarrow \text{identifier}, <\text{ident\_list}>$$

as

$$<\text{ident\_list}> \rightarrow \text{identifier} \mid \text{identifier}, <\text{ident\_list}>$$

- Another example:

$$<\text{stmt}> \rightarrow <\text{single\_stmt}> \mid \text{begin} <\text{stmt\_list}> \text{ end}$$

Lists

- Syntactic lists are described using recursion

$$<\text{ident\_list}> \rightarrow \text{ident} \mid \text{ident}, <\text{ident\_list}>$$

Derivation

- A repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)
  - Every string of symbols in a derivation is a sentential form
  - A sentence is a sentential form that has only terminal symbols
  - A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
  - A derivation may be neither leftmost nor rightmost

An Example of Derivation

- Given a grammar,

  <program> → <stmts>

  <stmts> → <stmt> | <stmt> ; <stmts>

  <stmt> → <var> = <expr>

  <var> → a | b | c | d

  <expr> → <term> + <term> | <term> - <term>

  <term> → <var> | const

- we can have the following derivation,

  <program> ⇒ <stmts> ⇒ <stmt> ⇒ <var> = <expr>

  ⇒ $a =$ <expr> ⇒ $a =$ <term> + <term>

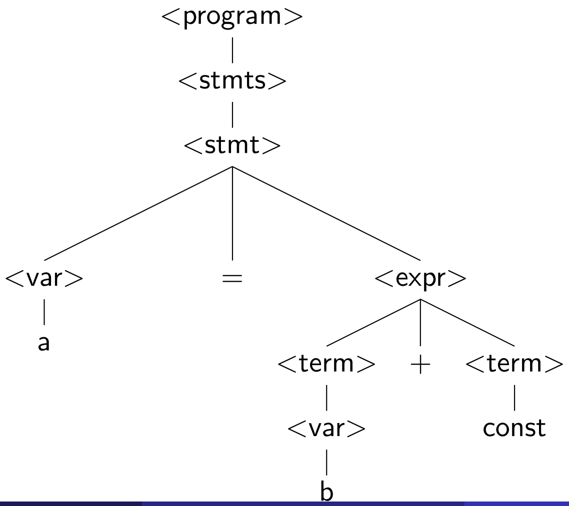
  ⇒ $a =$ <var> + <term>

  ⇒ $a = b +$ <term>

  ⇒ $a = b +$ const

Parse Tree

- A parse tree is a hierarchical representation of a derivation
- Example:



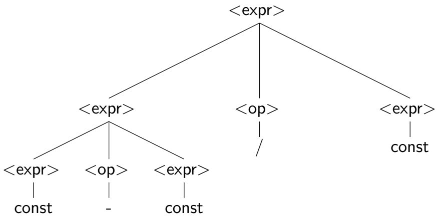
Ambiguity in Grammars

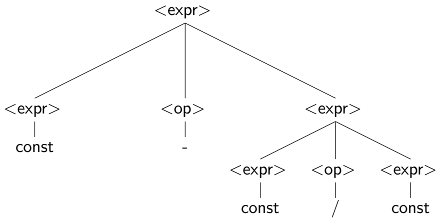
- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

Example of Ambiguous Grammar and Parse Trees

$$<expr> \rightarrow <expr> <op> <expr> \mid const$$

$$<op> \rightarrow / \quad \mid \quad -$$

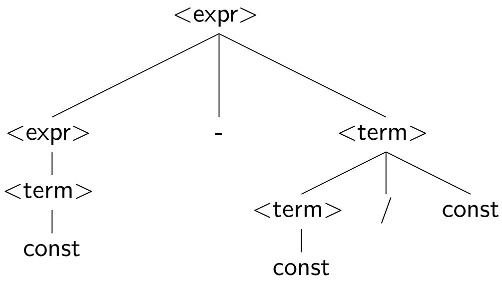




Unambiguous Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity
- Example:

$$
\begin{aligned}
<\text{expr}> &\rightarrow <\text{expr}> \; - <\text{term}> \;|<\text{term}> \\
<\text{term}> &\rightarrow <\text{term}> \; / \; \text{const} \;|\; \text{const}
\end{aligned}
$$



Associativity of Operators

- Operator associativity can also be indicated by a grammar
- Example: compare the following two grammars
  1. Ambiguous grammar

$$<\text{expr}> \ \rightarrow <\text{expr}> \ + <\text{expr}> \mid \text{const}$$

  2. Unambiguous grammar

$$<\text{expr}> \ \rightarrow <\text{expr}> \ + \ \text{const} \mid \text{const}$$

Extended BNF (EBNF)

- The extensions *do not* enhance the descriptive power of BNF; they only increase its *readability* and *writability*
- Optional parts are placed in brackets [ ], e.g.,

$$<\text{proc\_call}> \;\rightarrow\; \text{ident}\ [(<\text{expr\_list}>\ )]$$

- Alternative parts of RHSs are placed inside ( ) and separated via |, e.g.,

$$<\text{term}> \;\rightarrow <\text{term}>\ (+|-)\ \text{const}$$

- Repetitions (0 or more times) are placed inside {},

$$<\text{ident}> \;\rightarrow\; \text{letter}\ \{\ \text{letter} \mid \text{digit}\ \}$$

- Can you rewrite the above examples without using extensions?

Recent Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a : instead of $\rightarrow$
- Use of $_{\text{opt}}$ for optional parts
- Use of $_{\text{oneof}}$ for choices

Static Semantics

- Context-free grammars (CFGs) has limitations to describe the syntax of programming languages
  - Some are context-free, but cumbersome to be described in CFGs, e.g., type constraints
  - Some are non context-free, e.g., variables must be declared before they are used
- Static semantics rules: checking and analysis of the rules can be done at compile time

Attribute Grammar

- Formal approach both to describing and checking the correctness of the static semantics rules of a program
- Additions to CFGs to carry some semantic info on parse tree nodes
  - Static semantics specification
  - Static semantics checking

Definition of Attribute Grammar

- An attribute grammar is a context-free grammar $G = (S, N, T, P)$ with the following additions:
  - For each grammar symbol x there is a set $A(x)$ of attribute values
  - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of predicates to check for attribute consistency

Rules in Attribute Grammar

- Let $X_0 \to X_1 \ldots X_n$ be a rule
- Functions of the form $S(X_0) = f(A(X_1), \ldots, A(X_n))$ define synthesized attributes
- Functions of the form $I(X_j) = f(A(X_0), \ldots, A(X_n))$, for $i \leq j \leq n$, define inherited attributes
- Initially, there are intrinsic attributes on the leaves

An Example of Attribute Grammars

- Syntax

$$
\begin{aligned}
<\text{assign}> &\rightarrow <\text{var}> \; = <\text{expr}> \\
<\text{expr}> &\rightarrow <\text{var}> \; + <\text{var}> \; | <\text{var}> \\
<\text{var}> &\rightarrow A|B|C
\end{aligned}
$$

- actual_type: synthesized for <var> and <expr>
- expected_type: inherited for <expr>

An Example of Attribute Grammars

- Syntax rule:

$$<\text{expr}> \ \rightarrow <\text{var}> [1] + <\text{var}> [2]$$

  - Semantic rules:

$$<\text{expr}> .\text{actual\_type} \rightarrow <\text{var}> [1].\text{actual\_type}$$

  - Predicate:

$$<\text{var}> [1].\text{actual\_type} == <\text{var}> [2].\text{actual\_type}$$
$$<\text{expr}> .\text{expected\_type} == <\text{expr}> .\text{actual\_type}$$

- Syntax rule:

$$<\text{var}> \ \rightarrow \ \text{id}$$

  - Semantic rule:

$$<\text{var}> .\text{actual\_type} \leftarrow lookup(<\text{var}> .\text{string})$$

Compute Attribute Values

- If all attributes were *inherited*, the tree could be decorated in top-down order.
- If all attributes were *synthesized*, the tree could be decorated in bottom-up order.
- In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

An Example of Computing Attribute Values

<expr> .expected_type $\leftarrow$ inherited from parent

<var> $[1]$.actual_type $\leftarrow lookup(A)$
<var> $[2]$.actual_type $\leftarrow lookup(B)$

<var> $[1]$.actual_type $==$ <var> $[2]$.actual_type

<expr> .actual_type $\leftarrow$ <var> $[1]$.actual_type
<expr> $.actual\_type ==$ <expr> .expected_type

Dynamic Semantics

- meaning, of the expressions, statements, and program units of a programming language
- need for a methodology and notation for describing semantics.
  - Programmers need to know what statements mean
  - Compiler writers must know exactly what language constructs do
  - Correctness proofs would be possible
  - Compiler generators would be possible
  - Designers could detect ambiguities and inconsistencies

Describing Semantics

- no universally accepted notation or approach has been devised for dynamic semantics
- briefly describe several of the methods that have been developed
  - Operational Semantics
  - Denotational Semantics
  - Axiomatic Semantics

Operational Semantics

- To describe the meaning of a statement or program by specifying the effects of running it on a machine.
- The effects on the machine are viewed as the sequence of changes in its state (memory, registers, etc.)
- To use operational semantics for a high-level language, a *virtual machine* or an idealized computers is used

Applications of Operational Semantics

- A complete computer simulation
- The process:
  - Build a translator (translates source code to the machine code of an idealized computer)
  - Build a simulator for the idealized computer
- Evaluation of operational semantics:
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (e.g., VDL), it was used for describing semantics of PL/I.

Evaluation

- good if used informally (e.g., in programming language manuals)
- extremely complex if used formally (e.g.,VDL)

Denotational Semantics

- Originally developed in [Strachey and Scott, 1970, Scott and Strachey, 1971]
- The most rigorous and most widely known formal method for describing the meaning of programs
- Based on recursive function theory

Constructing Denotational Semantics Specification

- define *syntactic domain*: mathematical objects for language entities
- define *semantic domain*: function that maps language entities onto mathematical objects
- syntactic domain (domain $\mathbb{D}$): collection of values that are legitimate parameters to the function
- semantic domain (range $\mathbb{R}$): collection of objects to which the parameters are mapped

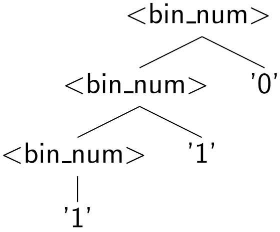
$$f : \mathbb{D} \mapsto \mathbb{R}$$

An Example: Binary Numbers

- Grammar for binary numbers

$$
\begin{aligned}
<\text{bin\_num}> \; &\rightarrow '0' \\
&|'1' \\
&|<\text{bin\_num}> \; '0' \\
&|<\text{bin\_num}> \; '1'
\end{aligned}
$$

- Parse Tree for 110



An Example: Binary Numbers

- Now need to define the *meaning* of binary numbers
- *syntactic domain* (domain):

$$\mathbb{D} = \{'0', '1', \text{<bin\_num>}\ '0', \text{<bin\_num>}\ '1'\}$$

- *semantic domain* (range):

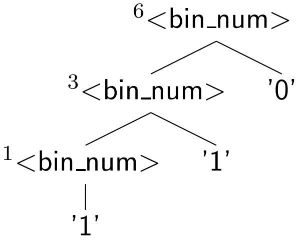
$$\mathbb{R} = \{0, 1, 2 \cdot M_{bin}(\text{<bin\_num>}\ ), 2 \cdot M_{bin}(\text{<bin\_num>}\ ) + 1\}$$

- mapping from domain onto range $M_{bin} : \mathbb{D} \mapsto \mathbb{R}$

$$M_{bin}('0') = 0$$
$$M_{bin}('1') = 1$$
$$M_{bin}(\text{<bin\_num>}\ '0') = 2 \cdot M_{bin}(\text{<bin\_num>}\ )$$
$$M_{bin}(\text{<bin\_num>}\ '1') = 2 \cdot M_{bin}(\text{<bin\_num>}\ ) + 1$$

An Example: Binary Numbers

- Decorated Parse Tree for 110



An Example: Decimal Numbers

- Grammar for decimal numbers (in EBNF)

$$
\begin{aligned}
\text{<dec\_num>} \rightarrow & '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9' \\
& |\text{<dec\_num>}\ ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')
\end{aligned}
$$

- What are the mapping function and its syntactic and semantic domains?
- Can you provide an example of decorated parse tree for 3231?

The State of a Program

- Let the state s of a program be represented as a set of ordered pairs

$$s = \{< i_1, v_1 >, < i_2, v_2 >, ..., < i_n, v_n >\}$$

  where
  i_j is the name of $j - th$ variable and v_j is the current value of variable i_j, $1 \leq j \leq n$. The value of v_j can be the special value *undef*, which indicates that its associated variable is currently undefined.
- Let $VARMAP$ be a function of two parameters: a variable name and the program state. The value of $VARMAP(i_j, s)$ is v_j (the value paired with i_j in state s).
- See the binary numbers and decimal numbers examples

Three Language Constructs

- Expressions
- Assignment Statements
- Logical Pretest Loops

Expressions

- Map expressions onto $\mathbb{Z} \cup \{error\}$
- Assume
  - Expressions have no side effects
  - Operators are $+$ and $*$
  - Expression can have at most one operator
  - Only operands are scalar integer variables and integer literals
  - There are no parentheses
  - The value of an expression is an integer.

Grammar of Expressions

- Grammar

$$
\begin{aligned}
<\text{expr}> &\rightarrow <\text{dec\_num}> |<\text{var}> |<\text{binary\_expr}> \\
<\text{binary\_expr}> &\rightarrow <\text{left\_expr}> <\text{operator}> <\text{right\_expr}> \\
<\text{left\_expr}> &\rightarrow <\text{dec\_num}> |<\text{var}> \\
<\text{right\_expr}> &\rightarrow <\text{dec\_num}> |<\text{var}> \\
<\text{operator}> &\rightarrow +|*
\end{aligned}
$$

Mapping Function for the Expressions

- use $\Delta =$ to define mathematical functions

```
M_e (<expr>, s) Δ= case <expr> of
                        <dec_num> => M_dec (<dec_num>, s)
                        <var> => if VARMAP (<var>, s) == undef
                                     then error
                                     else VARMAP (<var>, s)
                        <binary_expr> =>
                         if(M_e (<binary_expr>.<left_expr>,s) == undef OR
                           M_e (<binary_expr>.<right_expr>, s) == undef)
                          then error
                          else if (<binary_expr>.<operator> == '+')
                                then M_e (<binary_expr>.<left_expr>, s) +
                                     M_e (<binary_expr>.<right_expr>, s)
                                else M_e (<binary_expr>.<left_expr>, s) *
                                     M_e (<binary_expr>.<right_expr>, s)
```

Assignment Statement

- Maps state sets to state sets $s \cup \{error\}$

$$
\begin{aligned}
M_a(x = E, s) \;\Delta\!= \;& \text{if } M_e(E, s) == \textbf{error} \\
& \text{then } \textbf{error} \\
& \text{else } s' = \{<i_1, v_1'>, <i_2, v_2'>, \ldots, <i_n, v_n'>\}, \text{ where} \\
& \qquad \text{for } j = 1, 2, \ldots, n \\
& \qquad\; \text{if } i_j == x \\
& \qquad\quad \text{then } v_j' = M_e(E, s) \\
& \qquad\quad \text{else } v_j' = \text{VARMAP}(i_j, s)
\end{aligned}
$$

Logical Pretest Loops

- Maps state sets to state sets $s \cup \{error\}$

$$
\begin{aligned}
M_l(\texttt{while } B \texttt{ do } L, s) \;\Delta= \;& \text{if } M_b(B, s) == \textbf{undef} \\
& \text{then } \textbf{error} \\
& \text{else if } M_b(B, s) == \text{false} \\
& \qquad \text{then } s \\
& \qquad \text{else if } M_{sl}(L, s) == \textbf{error} \\
& \qquad\qquad \text{then } \textbf{error} \\
& \qquad\qquad \text{else } M_l(\texttt{while } B \texttt{ do } L, M_{sl}(L, s))
\end{aligned}
$$

Meaning of Loops

- The value of the program variables after the statements in the loop have been executed the prescribed number of times, assuming there have been no errors.
- The loop has been converted from *iteration* to *recursion*, where the recursion control is mathematically defined by other recursive state mapping functions
- *Recursion* is easier to describe with mathematical rigor than *iteration*.
- Observation: according to the defintion, like actual program loops, may compute nothing because of nontermination

Evaluation

- Can be used to prove the correctness of programs
- Provides a rigorous way to think about programs
- Can be an aid to language design
- Has been used in compiler generation systems
- Because of its complexity, it are of little use to language users

Axiomatic Semantics

- Based on formal logic (predicate calculus)
- Original purpose: formal program verification
- Axioms or inference rules are defined for each statement type in the language (to allow transformations of logic expressions into more formal logic expressions)
- The logic expressions are called *assertions*

Assertions in Axiomatic Semantics

- An assertion before a statement (a precondition) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a postcondition
- A weakest precondition is the least restrictive precondition that will guarantee the postcondition

Evaluation

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

Denotation and Operational Semantics

- In operational semantics, the state changes are defined by coded algorithms
- In denotational semantics, the state changes are defined by rigorous mathematical functions

Summary

- BNF and context-free grammars are equivalent meta-languages
  - Well-suited for describing the syntax of programming languages
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language
- Three primary methods of semantics description
  - Operation, axiomatic, denotational

References I

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2006).
Introduction to Automata Theory, Languages, and Computation (3rd Edition).
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Scott, D. S. and Strachey, C. (1971).
Toward a mathematical semantics for computer languages, volume 1.
Oxford University Computing Laboratory, Programming Research Group.

Strachey, C. and Scott, D. (1970).
Mathematical semantics for two simple languages.
Princeton Univ.