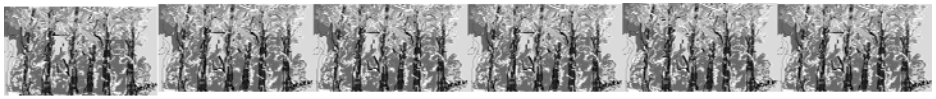


Chapter 3 – Syntax and Semantics
Introduction to Programming Languages
First Edition, 2013

Author: Arvind Bansal
Kent State University
Kent, OH 44242

Background Concepts

- Abstract concepts in computation (section 2.4)
  - Environment and store
- Control flow diagrams (chapter 1)
- von Neuman machine (section 2.1)
- Discrete structures (section 2.2)
  - Finite State machines (subsection 2.2.5)
  - Recursion (subsection 2.2.4)
- Data structures
  - Trees (subsection 2.3.5)
  - Graphs (subsection 2.3.6)

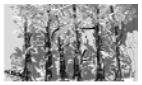
Topics Covered

- Introduction to syntax and semantics
- Grammars
- Syntax diagrams
- Validating sentence structure
- Different types of Semantics
- Summary

Introduction

- Two major components: syntax and semantics
- Syntax: validation of sentence structure
  - Has grammar rules, alphabets and validation that a sentence can be derived using grammar rules
  - Alphabet is the set of reserved words in a programming language
- Semantics: associating unique meaning to a sentence
  - Needs a semantic domain where meaning is defined
  - Semantic algebra describes the operations in semantic domain
  - Semantic rules describe the meaning of syntactic rules in the semantic domain
  - There are different types of semantics: operational semantics, axiomatic semantics, denotational semantics, action semantics, and behavior semantics
  - Example: meaning of 1011 in decimal number domain ➔ $1.2^3 + 0.2^2 + 1.2^1 + 1.2^0 = 11_{10}$

Syntax Grammar and Parse Tree



- A quadruple of the form (Σ, N, P, S)
  - Σ is alphabet. Terminal symbols are subset of alphabet and make a sentence.
  - N is set of nonterminal symbol,
  - P is set of production rules
  - S is start symbol
- A grammar rule has left hand and right hand side
  - LHS contains non-terminal symbol that is to be expanded
- Parse tree
  - Formed by matching right had side of rule and replacing by LHS until start symbol
  - Should be unique for a sentence

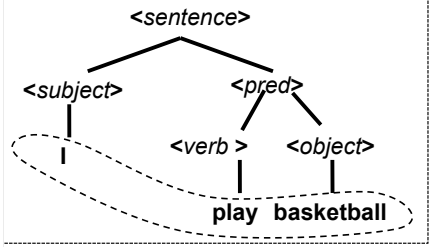
<*sentence*>::= <*subject*> <*pred*>

<*pred*> ::= <*verb*><*object*>

<*subject*> ::= **I | We | You**

<*verb*> ::= **play | dance**

<*object*> ::= **soccer| basketball**



Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

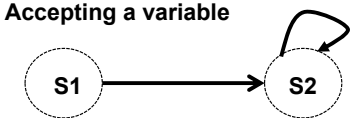
Types of Grammar - I

- Regular (type-3)
  - Describe nondeterministic FSA
  - Multiple states with multiple transitions between them
  - Applied in lexical analysis to accept a reserved word and generate a token
  - States as nonterminal symbols

<*S1*> → <*letter*> <*S2*>

<*S2*> → <*letter*> <*S2*> |
<*digit*> <*S2*> | '_' <*S2*> | ε

Accepting a variable



- Context free (type-2)
  - LHS is one nonterminal symbol; **no** terminal symbol on LHS
  - RHS is a combination of nonterminal and terminal symbols
  - Developing parsers for CFG is easy as nonterminal symbols can be expanded without any context of terminal symbol
  - Used for grammar of programming languages
  - more powerful than regular grammar
  - **Example:** <**S**> = **a** <*S*> **b** generates $\mathbf{a}^n\mathbf{b}^n$

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Context Sensitive Grammar

- Have a terminal symbol on the LHS side that provides the context for expanding nonterminal symbol
- Is more powerful than CFG (context free grammar)
- Very difficult to write parser software for CSG due to rule explosion. Can generate a string of the form $a^nb^nc^n$ not possible for CFG
- Example

  <*S*> ::= a<*S*>c | ε

  <*S*> c ::= b<*S*>cc
- Not used in the grammars for programming languages

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Backus Naur Form

- Used to represent context free grammar
- Nonterminal symbols are enclosed in angular brackets
- Multiple definitions are separated by vertical bar '|'
- Null statement is represented as ε
- Grammar uses multiple definitions, concatenation, tail-recursive definitions and recursive definitions

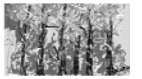
  <*statement-seq*> ::= <*statement*> '**;**' <*statement-seq*> | ε
- Limitations
  - New rules for multiple definitions causes explosion of rules.
  - Tail-recursive definitions for zero or more occurrences need an additional rule, and cause rule explosion.
  - Optional definition (0 or 1 occurrence) is handled using multiple definitions.

Introduction to Programming Languages, 1st edition, 2013, **ISBN**: 978-146-6565142

Grammar for Expression

<*expression*> ::= <*A-expr*> | <*L-expr*>
<*L-expr*> ::= true | false | <*identifier*> | not <*L-expr*> | <*comparison*> | <*L-expr*> <*L-op*> <*L-expr*>
<*comparison*> ::= <*A-expr*> <*comp-op*> <*A-expr*>
<*A-expr*> ::= <*number*> | <*identifier*> | <*A-expr*> <*A-op*> <*A-expr*>
<*identifier*> ::= <*letter*><*alphanumerics*> | <*letter*>
<*alphanumerics*> ::= ε | <*digit-or-letter*><*alphanumerics*>
<*digit-or-letter*> ::= <*digit*> | <*letter*>
<*number*> ::= <*digit*> | <*number*><*digit*>
<*digit*> ::= **'0' | '1' | … |'8' | '9'**
<*alphabet*> ::= '**a'| 'b' | … | 'z'| 'A' | 'B'| …| 'Z'**
<*A-op*> ::= **'+' | '-' | '/' | '*'**
<*L-op*> ::= **'&&' | '||'**
<*comp-op*>::= **'>' | '<' | '>=' | '=<' | ' =='**

Extended BNF



- Extended BNF removes the limitations of BNF
  - Additional rule for multiple definition is removed by embedding multiple definition within left and right parenthesis.
  - Additional rule for tail-recursive definition of zero or more occurrence is replaced by {… }* and one or more occurrence by { … }+.
  - Optional (0 or 1 occurrence) is represented by square bracket […].
  - Multiple alternative definitions are represented in parenthesis: (Alternative$_1$ | Alternative$_2$ | … | Alternative$_N$).

BNF representation	EBNF representation
<NT> ::= alt$_1$| alt$_2$	<NT> ::= (alt$_1$ | alt$_2$)
<NT> ::= ε | optional-feature	[optional-feature]
<NT> ::= symbol <NT> | symbol	<NT> ::= {symbol}+
<NT> ::= symbol <NT> | ε	<NT> ::= {symbol}*
<NT> ::= '0' | '1'| '2'| '3'	<NT> ::= '0' – '3'

EBNF for an Expression

- Rules for multi-definitions and tail recursive definitions have been removed and embedded.
- Multiple definition for ranges have been replaced by ranges in <*letter*> and <*digit*>.
- Rules for optional definitions have been removed.

<*expression*> **::=** <*A-expr*> | <*L-expr*>
<*L-expr*> **::=** [**not**] (**true** | **false** | <*identifier*> | <*comparison*>)
{ ('**&&**' | '**||**') [**not**] (**true** | **false** | <*identifier*> | <*comparison*>)}*
<*comparison*> **::=** <*A-expr*> ('**>**' | '**<**' | '**>=**' | '**=<**' | '**==**') <*A-expr*>
<*A-expr*> **::=** (<*number*>|<*identifier*>) {('**+**' | '**-**' | '*' | '*/*') (<*number*> | <*identifier*>)}*
<*number*> **::=** { <*digit*>}+
<*identifier*> **::=** <*letter*> {(<*letter*> | <*digit*>)}*
<*letter*> **::=** (**'a' – 'z'** | **'A' – 'Z'**)
<*digit*> **::= '0' – '9'**

EBNF for Iteration

<*iteration*> **::=** <*for-loop*> | <*while-loop*> | <*do-while-loop*> | <*iterators*>
<*for-loop*> **::= for** '(' <*identifier*> = <*expression*> *')'* '**;** '
<*identifier*> <*op*> <*expression*> *';'* <*expression*>')' <*block*>
<*while-loop*> **::= while** '**(**' <*L-expr*> '**)**' <*block*>
<*do-while-loop*> **::= do** <*block*> '(' <*L-expr*>')'
<*iterator*> **::= foreach** (<*identifier*> in *(<identifier*> | <*enumeration*>) <*block*>
<*block*> **::=** '**{**' { <*statement*> ';'}* '**}**' | <*statement*> ';'
<*statement*> **::=** <*assignment*> | <*if-then-else*> | <*iteration*>
<*assignment*> **::=** <*identifier*> '**=**' <*expression*>
<*enumeration*> **::=** '**{**' <entity> { ',' <entity>}* '**}**' | <*identifier*>
<*entity*> **::=** <*integer*> | <*float*> | <*string*>
<*identifier*> **::=** <*letter*> {(<*letter*> | <*digit*>)}*
<*string*> **::=** '"' {(<*alphabet*> | <*digit*>)}* '"'
<*letter*> **::= 'A' – ' Z' | 'a' – 'z'**
<*integer*> **::=** [('**+**' | '-')] { <*digit*> }+
<*digit*> **::= '0' – '9'**

Attribute Grammar

- Production rules can be associated with attributes for
  - Handling computer limitations, error handling, meaning for low level code generation and constraints specification
- Types of architectural limitations
  - Size of the word; maximum size of strings; maximum limit on the integer or floating point numbers

Production Rule: <*int*> ::= <*int*><*digit*> | <*digit*>
Attributes: value(<*int*>) > -2**31; value(<*int*>) < 2**31 - 1
value(<int>$_1$) = 10 * value(<int>$_2$) + value(<digit>)
length(<*int*>$_{1)}$ = length(<*int*>$_2$) + 1
Attributes: value(<*int*>) = value(<*digit*>)

Hyper-rules and Meta-definitions

- Hyper-rules capture the general pattern in multiple production rules
- Meta-definitions are multiple definitions when applied to hyper-rules give multiple specific production rules
- Advantages
  - Number of rules are reduced
  - Grammar becomes more expressive
- Example
  - Multiple rules represent sequence of statements, sequence of declarations, sequence of parameters.
  - **Hyper-rule:** <*sequence*> **:** <*definition*> '**;**' <*sequence*> | ε
  - **Meta-definition:** <*definition*> **::** <*statement*> | <*declaration*> | <*actual-parameter*> | <*formal-parameter*>

Abstract Syntax

- To understand the property of control and data abstraction, syntax rules are abstracted.
- Abstract language constructs
  - Program, block, iteration, command, assignment, expression, declaration, formal parameter, actual parameter, definition, label etc.
- Properties
  - Abstract syntax rules omit low level production rules and nonterminal symbols and low level symbols such as delimiters, white spaces literals etc.
  - Abstract syntax is concise, and are used for code generation
  - There may be some ambiguity is abstract parse tree due to omission of low level details

Example of Abstract Syntax - I

<*l-value*> **::=** <*identifier*> | <*identifier*>.<*l-value*> | <*l-value*>[<*expression*>]
<*declarations*> **::= var** <*identifier*> <*type-exp*> | <type-exp>[<numeral>] |
struct {<*type-exp*>} <*identifier*> |
void <*identifier*> (<*formal-parameters*>) |
<*identifier*> **function** (<*formal-parameters*>)
<*expressions*> **::=** <*literal*> | <*identifier*> | <*l-value*>| (<*expressions*>) |
<*expression*> <*op*> <*expressions*> | <*op*> <*expressions*>
<*actual-parameters*> **::=** <*identifier*> , <*actual-parameters*>
<*formal-parameter*> **::=** <*identifier*> <*identifier-seq*>'; '<*formal-parameter*>

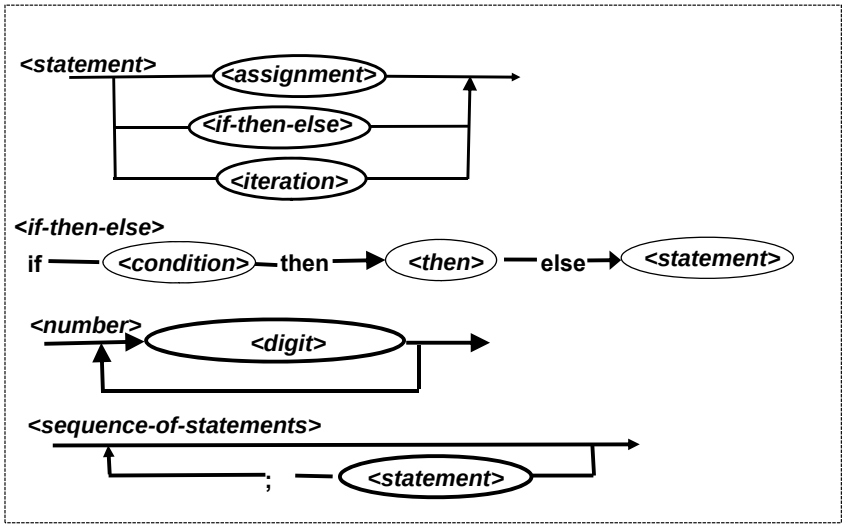
Example of Abstract Syntax - II

<*commands*> **::=** ‘{‘ <*commands*> ‘}’ | <*l-value*> ‘=‘ <*expression*> |
<*command*>’; ‘<*commands*> |
if <*expression*> **then** <*commands*> **else** <*commands*> |
if <*expression*> **then** <*commands*> |
while (<*expression*>) <*commands*> |
do <*commands*> (<*expression*>) |
for ‘(‘ <*l-value*> ‘=‘ <*expression*>’; ‘
<*expression*> ‘;’ <*expression*> ‘)’
<*commands*>
<*identifier*> ‘(‘…, <*expression*>, …’)’
<*sequencer*> **::= goto** <numeral>
<*program*> **::= main** <*identifier*>; <*declaration*> ; <*command*>

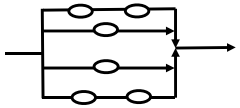
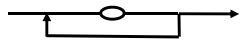
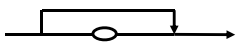
Syntax Diagrams

- Definition
  - Visual representation of production rules
  - Each diagram is a directed acyclic graph merging many production rules into one graph meaningfully
- Advantages
  - Visual representation is easy to comprehend by humans when number of rules are large
  - Combine multiple textual rules into one diagram based upon abstract meaning
- Translating Production rules to syntax diagrams
  - There is a input-end and an output-end
  - Multiple definitions becomes multiple forward paths
  - concatenation of symbols on RHS is represented as cascade
  - Tail recursive definitions are represented as feedback loop
  - Null definition is represented as a straight arrow

Example of Syntax Diagram



Production Rules Correspondence

Component	Syntax diagram
Concatenation	
Multiple definitions in BNF or grouping in EBNF shown as parallel forward branches	
Tail-recursive definition for one or more occurrence shown as feedback loop	
Tail-recursive definition for zero or more occurrence: entity in feedback loop	
Empty symbol: straight arrow	
Optional in EBNF: a forward branch for definition and an empty branch	

Translating Production Rules - I

- Grammar
  - Grouped into language relevant specific units that correspond to control or data abstractions.
- Translation
  - Top level rules are translated first,
  - Nonterminal symbols in the corresponding syntax diagrams are expanded further
- Example (see the syntax diagram in the next slide)

  <identifier> ::= *<letter>* {(*<letter>* | *<digit>*)}*

  <alphabet> = 'A' – ' Z' | 'a' – 'z'

  <digit> = '0' – '9'

Translating Production Rules II

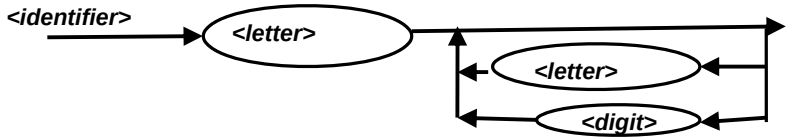


Figure 3.7a - Syntax diagram for production rule # 1 for <identifier>

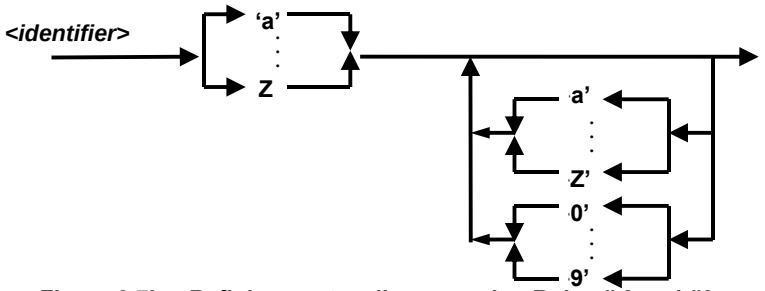


Figure 3.7b – Refining syntax diagram using Rules # 2 and #3

Translating Syntax Diagram

- Translating syntax diagrams
  - Reverse of making syntax diagrams
- Technique
  1. Take a syntax diagram that does not use any existing syntax diagram in its definition, and convert the inner most part to a production rule.
  2. If the production rule is already existing then use the existing production rule.
  3. Replace the inner most part by the nonterminal symbol of the new production rule, and modify the syntax diagram
  4. Repeat the process on the modified syntax diagram until a single nonterminal symbol is left
  5. Repeat this process for all the syntax diagrams

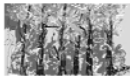
Validating Sentence Structure

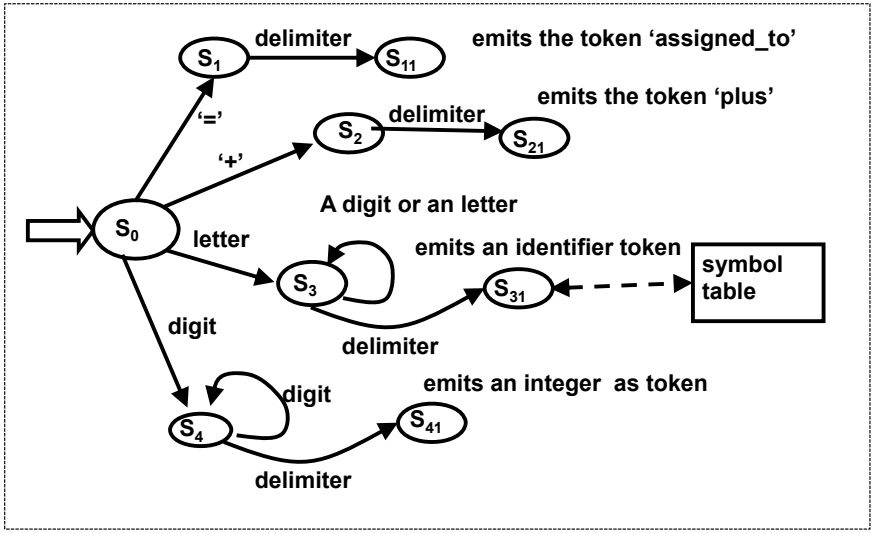
- Two step process: 1) Lexical analysis, 2) Parser
- Step 1 - Lexical analysis
  - Identifies the reserved words in the program, converts into a token – internalized representation for fast processing
  - Sends the tokenized stream to parser
- Step 2 – Parser
  - Takes the tokenized program, and using the grammar rules, generates a unique parse tree. This parse tree is used for code generation.
  - Parser is automatically generated using parser generators that takes a grammar input and generates a parser. There are many types of parsers such as LR (k) parsers, LL parsers

Lexical Analysis

- First phase of compilation
  - Modeled using a regular grammar and finite state automata
  - Identifies reserved words, identifiers and literals and converts them to tokens
  - Identifiers are stored in symbol table so that same token can be retrieved in future for the same identifier
- Process
  - Start from the initial state
  - Use the next character to move to the next state using a lookup table
  - Look ahead to resolve the ambiguity
  - Stop when a delimiter / end of line / end of file is found,
  - Emit the corresponding token

Lexical Analysis Example

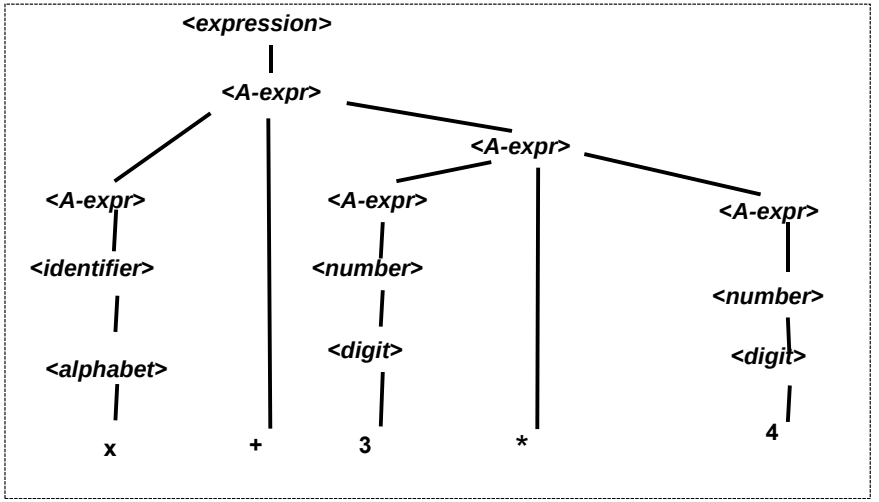




Parsing

- Parsing is a sentence validation process
- Takes the tokenized program stream as input
- Uses a language grammar and parser to parse
- Generates a tree for code generation process
- Process
  - Start with a given sentence
  - Find out a substring in the sentence that matches the right hand side of a production rule
  - Replace the substring by the left hand side non terminal
  - Repeat the process until start symbol is reached or no more reduction is possible
  - If the left over is not the start symbol then the sentence is not a valid sentence

Parse Tree for Expression



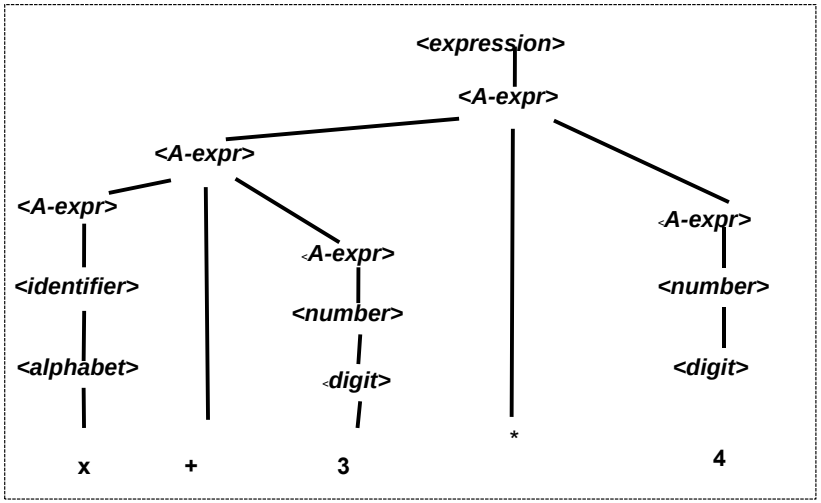
Simplified Parsing Algorithm

Algorithm bottom-up-parse-sentence;
Input: 1. A set of production rules R = {p_1, … p_n} of the grammar;
2. A sentence as a sequence of symbols S = s_0, …, s_m;
3. The start symbol *root*;
Output: A parse tree T;
{ reduced-form = S; parsing-error = false; T = null-tree;
while ((reduced-form ≠ root) || not(parsing-error))
{ **If** there exists a subsequence s_i .. s_j in *reduce*d-form such that
s_i… s_j == right-hand-side(p_i ∈ R) **where** 1 ≤ i ≤ n **{**
nonterminal = left-hand-side(p_i);
reduced-form = substitute(reduced-form, s_i… s_j, nonterminal);
T = T + edge(s_i ..s_j → left-hand-side(p_i));**}**
else parsing-error = true;**}**
If not(parsing-error) return(T); **else** print('parsing-error');
}

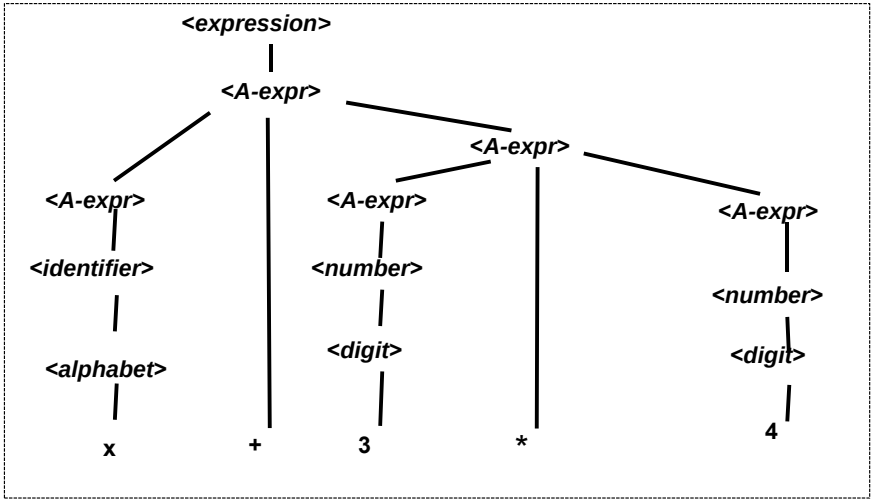
Grammar Ambiguity

- A grammar is ambiguous if
  - There are more than one parse tree for at least one sentence accepted by the grammar
  - Two parse trees means two different meanings for the same sentence – a violation of principle of programming language
- Cause of ambiguity
  - Multiple different rules merged into one multi definition rule
- Example of ambiguous grammar
  - Arithmetic expressions where addition/subtraction are treated equally multiplication/division in the same multi-definition rule
  - Nested if-then-else where pairing of multiple ifs is unclear with the corresponding else in multi-definition rule
- Handling
  - Stratify the rules with highest priority rule used first in parsing
  - **Example**: one multi-definition rule in arithmetic expression is split in multiple production rules: each rule has equal priority definitions.

Incorrect Parse Tree



Correct Parse Tree

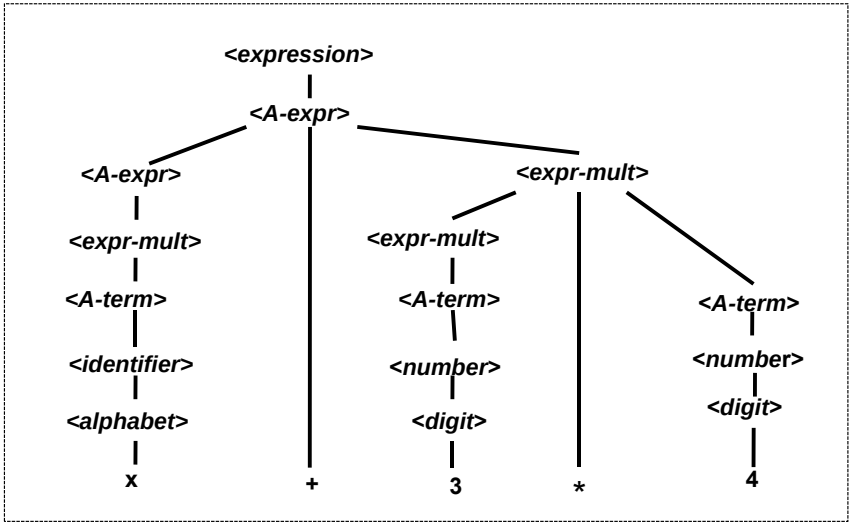


Unambiguous Grammar for Expression

- Production rules 2, 3 and 4 have been separated based upon priority of the arithmetic-operators
- Rules 5, 6 and 7 are separated based upon priorities of logical operators.

<*expression*> **::=** <*A-expr*> | <*L-expr*>	(1)
<A-expr> **::=** <*A-expr*> (‘**+**’ | ‘**-**‘)<*expr-mult*> | <*expr-mult*>	(2)
<*expr-mult*> **::=** <*expr-mult*> (‘*****’ | ‘*/*’) <*A-term*> | <*A-term*>	(3)
<*A-term*> **::=** ‘**(**‘ <*A-expr*> ‘**)**’ | <*identifier*> | <*number*>	(4)
<*L-expr*> **::=** <*L-expr*> ‘**||**’ <*expr-and*>	(5)
<*expr-and*> **::=** <*expr-and*> **‘&&’** <*L-term*>	(6)
<*L-term*> **::=** [**not**] (‘**(**‘ <*compare*> ‘**)**’ | ‘(‘ <*L-expr*> ‘)’ | <*identifier*> | **true** | **false**	(7)
<*compare*> **::=** <*A-expr*> (‘**>**’ | ‘**<**’ | ‘**>=**’ | ‘**=<**’ | ‘**==**’) <*A-expr*>	(8)
<*identifier*> **::=** <*letter*>*{(*<*letter*>|<*digit*>*)}**	(9)
<*number*> **::=** [(‘**+**’|’**-**‘)] {<*digit*>}+	(10)
<*letter*> **::=** ‘**a**’| ‘**b**’ | **…** |’**z**’| ‘**A**’ | ‘**B**’ | **…** | ‘**Z**’	(11)
<*digit*> **::=** ‘**0**’ | ‘**1**’| **…** |’**9**’	(12)

Parsing using Unambiguous Grammar



Ambiguities in Nested Structure

- Condition
  - Nested if-then-else statement has unequal number of if and else statement
  - else-statements have to be correctly matched
- Ambiguous grammar

  <*if-then-else-statement*> **::= if** <*condition*> **then** <*statement*>
  else <statement>

  <*statement*> ::= <*assignment*> | <*iteration*> | <*if-then-else*> | …

incorrect interpretation	Correct interpretation
if (x > 4) **then** **if** (y > 0) **then** return(1); **else** return(0)	**if** (x > 4) **then** **If** (y > 0) **then** return(1); **else** return(0);

Unambiguous Grammar for if-then-else

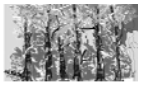
- The rule has been stratified into three different rules
  - <matched if-then-else> is tried in then part of <unmatched-if-then-else – statement>

<*if-then-else-statement*> ::= <*matched-if-then-else*> |
<*unmatched-if-then-else*>

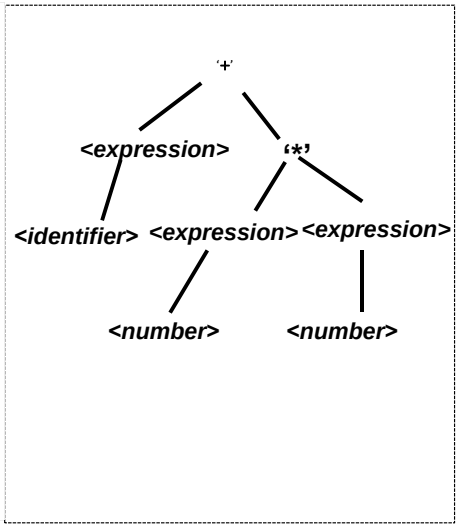
<*matched-if-then-else*> ::= **if** <*cond*> **then** <*matched-if-then-else*>
else <*matched-if-then-else*>|
<*other-statements*>

<*unmatched-if-then-else*> ::= **if** <*cond*> **then** <*if-then-else-statement*> |
if <*cond*> **then** <*matched-if-then-else*> **else**
<*unmatched-if-then-else*>

Abstract Syntax Tree



- Free of all the non-terminal symbols in the concrete syntax rules that are not in the abstract syntax rules
- Expressions are rooted at the operators
- Concrete syntax to abstract syntax conversion
  - Remove low level nonterminal symbols not contributing to meaning
  - Remove low level production rules
  - Remove delimiters and white spaces



Automated Parsing

- Top down or Bottom up
- Top down parsing (Recursive descent parsing)
  - Left most nonterminal symbol is expanded and matched with the corresponding part of a sentence
  - In case of mismatch backtracking is used that is inefficient
  - Predictive parsers improve efficiency and use an M X N table where M is number of nonterminal symbols and N is number of terminal symbols. Each cell of the table contains a production rule that matches
- Bottom up Parsing (Shift Reduce Parsing)
  - Starts parsing from the leaf node and moves up
  - Parses from the rightmost part of the sentence
  - LR(K) or LALR (Lookahead LR) parsers use K symbol look ahead to identify the production rules
  - Advantages are: (1) non-recursive; 2) non-backtracking; 3) can parse all programming constructs; and 4) identify grammar ambiguities

Operational Semantics I

- Modeled as the transition from one state to another state
  - State is a triple of the form (environment, store, dump)
  - Declaration changes environment
  - Assignment changes store
  - Procedure-call and return from procedure changes dump
- Small-step (or structural) operational semantics
  - Uses low level abstract instructions to explain semantics
  - Example: evaluating a literal, looking up an identifier, assignment, declaring a new identifier
- Big-step (natural semantics)
  - Used for transition of computation states for high level constructs such as for-loop, if-then-else statement, while-loop, assignment statement etc.

Small-step → Big-step Semantics

- Big-step operational semantics
  - Modeling higher level programming constructs as composition of low level programming constructs using control flow diagrams and small-step operational semantics
- Small-step operational semantics

(literal, σ) → literal

(identifier, σ) → r-value(identifier) (σ)
if ((identifier → l-value) ∈ σ^E and (l-value, r-value) ∈ σ^S

(*new* identifier, <σ^E, σ^S, σ^D>) → <$\sigma^E \oplus$ (identifier $\mapsto$ l-value), $\sigma^S \oplus$ (l-value $\mapsto$ undefined), σ^D>

(exp_1 op exp_2, σ) → $value_1$ op $value_2$ and σ does not alter
where (exp_1, σ) → $value_1$ and
(exp_2, σ) → $value_2$, and
op ∈ {add, subtract, multiply, divide}

(identifier = exp, <σ^E, σ^S, σ^D>) →
<σ^E, $\sigma^S \oplus$ (l-value(identifier) $\mapsto$ value, σ^D>
where (exp, <σ^E, σ^S, σ^D>) → value

Axiomatic Semantics - I

- Uses predicate calculus to express computation-state independent of any computer architecture such as von-Neuman machine
  - Computation-state is expressed as a combination of Boolean predicates
  - Assignment statement changes / inserts / deletes a Boolean expression
- Example 1: A = 5; B = 4; A = B + 4;

Pre-condition	Statement	Post-condition
undefined	A = 5	A == 5
A == 5	B = 4	(A == 5) and (B == 4)
(A == 5) and (B == 4)	A = B + 4	(A == B + 4) and (B == 4)

- Example 2: If (x > y) max = x else max = y;

Pre-condition	post-condition
<*Cond1*>	<*Cond1*> **and** ((x > y **and** max == x) **OR** (X =< y **and** max = y))

Axiomatic Semantics - II

- Handling Iteration
  - Iteration requires going through the loop over-and-over again
  - At the beginning of handling iteration initial condition is true
  - At the end of the iteration final condition becomes true
  - During the iteration, an invariant condition has to be true that can model every iteration cycle
  - An invariant condition is independent of the count of the iterative cycle
  - {I ⋀ B} **while** *B S;* {I ⋀ ¬ B}. **where** {I} S {I} % I is invariant
- Advantages of Axiomatic Semantics
  - Used to reason about final condition without executing a program
  - Used to reason about the correctness of the program
  - Used to construct a program using backward reasoning
- **Problem: For large programs reasoning is costly**

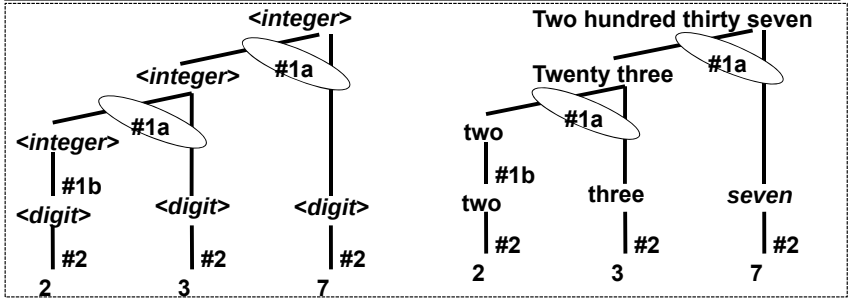
Denotational Semantics I

- Syntax rule is mapped into a semantic domain with one-to-one correspondence between syntax rule and semantic rule
- Components of Denotational semantics
  - Define semantic domain, semantic algebra and semantic rules
  - Every syntax rule has the corresponding semantic rule
  - Same parse tree is used to derive the meaning of a sentence by substituting a syntax rule by the corresponding semantic rule
- Denotational semantics vs. operational semantics

Denotational Semantics	Operational Semantics
1. Denotational semantics uses mathematical functions on syntax rules in a semantic domain 2. It is independent of computational state and abstract machine	1. Operational semantics uses state transition on computational states using an abstract machine such as von-Neuman machine. 2. It has no notion of semantic domain.

Denotational Semantics II

- **Syntax Rules:** <*integer*> ::= <*integer*> <*digit*> | <*digit*> (1)
  <*digit*> ::= '**0**' | '**1**'| … | '**9**' (2)
- **Semantic domain**: Base 10
- **Semantic Algebra**: **+, * :** $\mathbb{Z}_{10} \times \mathbb{Z}_{10} \rightarrow \mathbb{Z}_{10}$
- **Semantic Rules:** m(<*integer*> = 10 * m(<*integer*> + m(<*digit*> (1a)
  m(<*integer*>) = m(<*digit*>) (1b)
  m(<*digit*>) = **zero | one | … | nine** (2)



Decimal Number Syntax Rules

Rule # 1: *<number> ::= <integer> | <real>*

Rule # 2: *<integer> ::= <sign> <whole-number> | <whole-number>*

Rule # 3: *<real> ::= <sign> <unsigned-real> | <unsigned-real>*

Rule # 4: *<unsigned-real> ::= <whole-number> '.' <float> |*
<whole-number> '.' <float> 'E' *<integer>*

Rule # 5: *<whole-number>::=<whole-number><digit>|<digit>*

Rule # 6: *<float> :: <digit><float> | <digit>*

Rule # 7: *<digit>* ::= '0' | '1' | '2' | … | '9'

Rule # 8: *<sign>* = '+' | '−'

Decimal Number Semantics I

Domain:

Real number domain $\mathbb{R}_{10}$
Number domain: $\mathbb{N}_{10} = \mathbb{Z}_{10} \oplus \mathbb{R}_{10}$ % $\oplus$ denotes disjoint-union

Semantic Algebra:

In $\mathbb{R}_{10}$: real-add '$+^R$'; real-multiply '$\times^R$' : $\mathbb{R}_{10} \times \mathbb{R}_{10} \rightarrow \mathbb{R}_{10}$
In $\mathbb{Z}_{10}$: int-add '$+^I$'; int-multiply '$\times^I$' and '^' : $\mathbb{Z}_{10} \times \mathbb{Z}_{10} \rightarrow \mathbb{Z}_{10}$
In mixed domain: exponent '^': $(\mathbb{R}_{10} \times \mathbb{Z}_{10}) \rightarrow \mathbb{R}_{10}$

Semantic functions: ň; ř; ĭ;

Rule # 1: ň(*<number>*) ::= ĭ(*<integer>*) | ř(*<real>*)
Rule # 2: ĭ(*<integer>*) ::= ĭ(*<sign>*) $\times^I$ ĭ (*<whole-number>*) | ĭ(*<whole-number>*)
Rule # 3: ř (*<real>*) ::= ř(*<sign>*) $\times^R$ ř(*<unsigned-real>*) | ř(*<unsigned-real>*)

Decimal Number Semantics II

Rule # 4: ř (*<unsigned-real>*) ::= ř(*<whole-part>*) $+^R$ ř(*<decimal-part>*) |
(ř(*<whole-part>*) $+^R$ ř(*<decimal-part>*)) $\times$(10.0 ^ ĭ(*<integer>*))

Rule # 5a: ř(*<whole-part>*$_1$) ::= ř(*<whole-part>*$_2$) $\times^I$ 10.0 $+^R$ ř(*<digit>*) |
ř(*<digit>*)

Rule # 5b: ĭ(*<whole-number>*$_1$) ::= ĭ (*<whole-number>*$_2$) $\times^I$ ten $+^I$ ĭ(*<digit>*) |
ĭ (*<digit>*)

Rule # 6: ř(*<decimal-part>*$_1$) :: ř(*<digit>*) $+^R$ ř(*<decimal-part>*$_2$)) / 10.0 |
ř(*<digit>*) / 10.0

Rule # 7a: ř(*<digit>*) ::= float zero | float one | … | float nine % float 1 is 1.0

Rule # 7b: ĭ(*<digit>*) ::= zero | one | two | … | nine % interpretation of digits

Rule # 8a: ř(*<sign>*) ::= plus float-one | minus float-one % + 1.0 or -1.0

Rule # 8b: ĭ(*<sign>*) ::= plus one | minus one

Action Semantics

- Integrates the advantages of denotational semantics, axiomatic semantics, and operational semantics
- Explains the meaning using rules in natural English.
- Components: *action, data* and *yielders*
  - Action are the rules in natural English that explain control and data abstraction. Action could be *completing*, *diverging* or *fail*.
  - Two types of actions: primitive and combinatory. Primitive action is single step, and combinatory is composite action.
  - Information processed by an action is called data
  - Yielders are used to retrieve information after an action
- Advantages
  - Defines the meaning of programming languages comprehensibly
  - Uses a pragmatic approach to integrate all three semantics.

Specification of Action Semantics

- Notations
  - Specification of nodes to be constructed in the abstract syntax tree uses the notation [[...]]
  - Components are grouped as sequence of statements.
- Example
  - **execute** [[<*identifier*> '=' <*expression*>]]
  - **Action:** evaluate <*expression*> giving value then store the value in the l-value(<*identifier*>)

Similarly, the action semantics of the if-then-else statement is given by

- **execute** [[**'if'** <*expression*> '**then**' <*statement*>$_1$ '**else**' <*statement*>$_2$]]
- **Action:** evaluate <expression> giving truth value B then
  ((check truth-value of B **and** execute <statement>$_1$)
  (check not truth-value of B **and** execute <statement>$_2$))

Behavioral Semantics

- Used for behavioral model of object oriented languages and domain specific languages
  - Overall system is a network of interacting objects
  - Overall state is a cumulative system state of all active objects
  - Reaction to messages is a means of transition of the system state
- Modeling a state
  - Each class of object is modeled as n-tuple: (attributes, methods, messages received, messages emitted, transitions in response to input messages, and triggers that initiate a reaction)
  - System state is a collection of states of individual objects

Summary - I

- There are two major components: syntax and semantics
  - Syntax validates the sentence structure according to a grammar
  - Semantics provides the meaning to a sentence
- A grammar is a quadruple { <*Start symbol*>, <*Nonterminal symbols*>, <Alphabet>, <Production rules>}
  - Grammar has to be unambiguous for the unique meaning of a sentence
  - Programming languages use two types of grammars: regular grammar for lexical analysis and context free grammar for parsing sentences
  - Regular grammar is based on FSA, and lexical analysis generates a tokenized stream to be parsed
  - Context free grammar is represented using BNF and EBNF
  - Attribute grammar augments CFG with attributes that can be restrictions imposed by architecture or designers or meaning associated with the rule for code generation
- Humans understand syntax diagrams better

Summary - II

- Syntax Rule to Syntax Diagram Conversion
  - Group rules based upon control and data abstractions
  - Use correspondence between syntax rule and syntax diagrams to form a composite syntax diagram for each group
- Syntax Diagram to Syntax Rule Conversion
  - Replace the innermost part by a syntax rule, and substitute that part by a non-terminal symbol unless only one nonterminal symbol is left
- There are five different types of semantics
  - Operational semantics describes the meaning as transition from computational states described as triple (environment, store, dump)
  - Axiomatic semantics describes computational state as a combination of Boolean predicates and is independent of any architecture
  - Denotational semantics describes semantics as one to one correspondence between syntax rules and semantic rules in a domain
  - Action semantics is a more comprehensible integration of above three
  - Behavior semantics describes state as cumulative sum of object-states