Language Translation Principles

...or, how to explain C++ to a linguist
Attributes of a language

• Syntax: rules describing use of language tokens
• Semantics: logical meaning of combinations of tokens
• In a programming language, “tokens” include identifiers, keywords, and punctuation
Linguistic correctness

• A **syntactically correct** program is one in which the tokens are arranged so that the code can be successfully translated into a lower-level language

• A **semantically correct** program is one that produces correct results
Language translation tools

• Parser: scans source code, compares with established syntax rules
• Code generator: replaces high level source code with semantically equivalent low level code
Techniques to describe syntax of a language

- Grammars: specify how you combine atomic elements of language (characters) to form legal strings (words, sentences)
- Finite State Machines: specify syntax of a language through a series of interconnected diagrams
- Regular Expressions: symbolic representation of patterns describing strings; applications include forming search queries as well as language specification
Elements of a Language

- Alphabet: finite, non-empty set of characters
  - not precisely the same thing we mean when we speak of natural language alphabet
  - for example, the alphabet of C++ includes the uppercase and lowercase letters of the English alphabet, the digits 0-9, and the following punctuation symbols: 
    `{,},[,],(),+,,-,*,/,%,=,>,<,!,&,|,’,”,,,,:,:,\`
  - Pep/8 alphabet is similar, but uses less punctuation
  - Language of real numbers has its own alphabet; the set of characters `{0,1,2,3,4,5,6,7,8,9,+,,-,.}`
Language as ADT

• A language is an example of an Abstract Data Type (ADT)

• An ADT has these characteristics:
  – Set of possible values (an alphabet)
  – Set of operations on those values

• One of the operations on the set of values in a language is concatenation
Concatenation

• Concatenation is the joining of two or more characters to form a string

• Many programming language tokens are formed this way; for example:
  – > and = form >=
  – & and & form &&
  – 1, 2, 3 and 4 form 1234

• Concatenation always involves two operands – either one can be a string or a single character
String characteristics

• The number of characters in a string is the string’s length

• An empty string is a string with length 0; we denote the empty string with the symbol $\varepsilon$

• The $\varepsilon$ is the identity element for concatenation; if $x$ is string, then:
  $$\varepsilon x = x \varepsilon = x$$
Closure of an alphabet

• The set of all possible strings that can be formed by concatenating elements from an alphabet is the alphabet’s closure, denoted $T^*$ for some alphabet $T$

• The closure of an alphabet includes strings that are not valid tokens in the language; it is not a finite set

• For example, if $R$ is the real number alphabet, then $R^*$ includes:
  -0.092 and 563.18 but also .0.0.- and 2-4-2.9..-5.
Languages & Grammars

• A language is a subset of the closure of an alphabet
• A grammar specifies how to concatenate symbols from an alphabet to form legal strings in a language
Parts of a grammar

• N: a nonterminal alphabet; each element of N represents a group of characters from:
  – T: a terminal alphabet
  – P: a set of rules for string production; uses nonterminals to describe language structure
  – S: the start symbol, an element of N
Terminal vs. non-terminal symbols

- A non-terminal symbol is used to describe or represent a set of terminal symbols.
- For example, the following standard data types are terminal symbols in C++ and Java: int, double, float, char.
- The non-terminal symbol <type-specifier> could be used to represent any or all of these.
Valid strings

- S (the start symbol) is a single symbol, not a set
- Given S and P (rules for production), you can decide whether a set of symbols is a valid string in the language
- Conversely, starting from S, if you can generate a string of terminal symbols using P, you can create a valid string
Productions

A → w

a non-terminal produces a string of terminals & non-terminals
Derivations

• A grammar specifies a language through the derivation process:
  – begin with the start symbol
  – substitute for non-terminals using rules of production until you get a string of terminals
Example: a grammar for identifiers (a toy example)

- \( N = \{<\text{identifier}>, <\text{letter}>, <\text{digit}>\} \)
- \( T = \{a, b, c, 1, 2, 3\} \)
- \( P = \) the productions: (\( \rightarrow \) means “produces”)
  1. \(<\text{identifier}> \rightarrow <\text{letter}>\)
  2. \(<\text{identifier}> \rightarrow <\text{identifier}><\text{letter}>\)
  3. \(<\text{identifier}> \rightarrow <\text{identifier}><\text{digit}>\)
  4. \(<\text{letter}> \rightarrow a\)
  5. \(<\text{letter}> \rightarrow b\)
  6. \(<\text{letter}> \rightarrow c\)
  7. \(<\text{digit}> \rightarrow 1\)
  8. \(<\text{digit}> \rightarrow 2\)
  9. \(<\text{digit}> \rightarrow 3\)
- \( S = <\text{identifier}> \)
Example: deriving a12bc:

\[
\begin{align*}
<\text{identifier}> & \Rightarrow <\text{identifier}>\langle\text{letter}\rangle \text{ (rule 2)} \\
& \Rightarrow <\text{identifier}>c \text{ (rule 6)} \\
\Rightarrow \text{ means} & \Rightarrow <\text{identifier}>\langle\text{letter}\rangle c \text{ (rule 2)} \\
\Rightarrow \text{ derives in one} & \Rightarrow <\text{identifier}>bc \text{ (rule 5)} \\
\text{ step} & \Rightarrow <\text{identifier}>\langle\text{digit}\rangle bc \text{ (rule 3)} \\
& \Rightarrow <\text{identifier}>2bc \text{ (rule 8)} \\
& \Rightarrow <\text{identifier}>\langle\text{digit}\rangle 2bc \text{ (rule 3)} \\
& \Rightarrow <\text{identifier}>12bc \text{ (rule 7)} \\
& \Rightarrow <\text{letter}>12bc \\
& \Rightarrow \text{a12bc}
\end{align*}
\]
Closure of derivation

• The symbol $\Rightarrow^*$ means “derives in 0 or more steps”

• A language specified by a grammar consists of all strings derivable from the start symbol using the rules of production
  – provides operational test for membership in the language
  – if a string can’t be derived using production rules, it isn’t in the language
Example: attempting to derive 2a

\[
\text{<identifier>} \Rightarrow \text{<identifier><letter>}
\]
\[\Rightarrow \text{<identifier>}a\]

• Since there is no \text{<identifier>} \rightarrow \text{<digit>}
  combination in the production rules, we can’t proceed any further

• This means that 2a isn’t a valid string in our language
A grammar for signed integers

- $N = \{I, F, M\}$
  - $I$ means integer
  - $F$ means first symbol; optional sign
  - $M$ means magnitude

- $T = \{+,-,d\}$ ($d$ means digit 0-9)

- $P =$ the productions:
  1. $I \rightarrow FM$
  2. $F \rightarrow +$
  3. $F \rightarrow -$  
  4. $F \rightarrow \varepsilon$ (means +/- is optional)
  5. $M \rightarrow dM$
  6. $M \rightarrow d$

- $S = I$
Examples

• Deriving 14:

\[ I \Rightarrow FM \Rightarrow \epsilon M \Rightarrow dM \Rightarrow dd \Rightarrow 14 \]

• Deriving -7:

\[ I \Rightarrow FM \Rightarrow -M \Rightarrow -d \Rightarrow -7 \]
Recursive rules

• Both of the previous examples (identifiers, integers) have rules in which a nonterminal is defined in terms of itself:
  – `<identifier> → <identifier><letter>` and
  – `M → dM`

• Such rules produce languages with infinite sets of legal sentences
Context-sensitive grammar

- A grammar in which the production rules may contain more than one non-terminal on the left side.
- The opposite (all of the examples we have seen thus far), have production rules restricted to a single non-terminal on the left: these are known as context-free grammars.
Example

- $N = \{A,B,C\}$
- $T = \{a,b,c\}$
- $P$ = the productions:
  1. $A \rightarrow aABC$
  2. $A \rightarrow abC$
  3. $CB \rightarrow BC$
  4. $bB \rightarrow bb$
  5. $bC \rightarrow bc$  
    This rule is context-sensitive: $C$ can be substituted with $c$ only if $C$ is immediately preceded by $b$
  6. $cC \rightarrow cc$

- $S = A$
Context-sensitive grammar

- \( N = \{A, B, C\} \)
- \( T = \{a, b, c\} \)
- \( P = \) the productions
  1. \( A \rightarrow aABC \)
  2. \( A \rightarrow abC \)
  3. \( CB \rightarrow BC \)
  4. \( bB \rightarrow bb \)
  5. \( bC \rightarrow bc \)
  6. \( cC \rightarrow cc \)
- \( S = A \)

Example:

aaabbbcccc is a valid string by:

\[
\begin{align*}
A & \rightarrow aABC (1) \\
& \rightarrow aaABCBC (1) \\
& \rightarrow aaabCBCBC (2) \\
& \rightarrow aaabBCCBC (3) \\
& \rightarrow aaabBCBCC (3) \\
& \rightarrow aaabBBCCC (3) \\
& \rightarrow aaabbBCCC (4) \\
& \rightarrow aaabbbCCC (5) \\
& \rightarrow aaabbbC (6) \\
& \rightarrow aaabbbccc (6)
\end{align*}
\]

Here, we substituted \( c \) for \( C \); this is allowable only if \( C \) has \( b \) in front of it.

\[
\begin{align*}
& \rightarrow aaabbbbcC (5) \\
& \rightarrow aaabbbcccC (6) \\
& \rightarrow aaabbbccc (6)
\end{align*}
\]
Valid & invalid strings from previous example:

- Valid:
  - abc
  - aabbcc
  - aabc

- Invalid:
  - cba
  - bbbccc
  - aaac

The grammar describes a language consisting of strings that start with a number of a’s, followed by an equal number of b’s and c’s; this language can be defined mathematically as:

\[ L = \{a^n b^n c^n \mid n > 0\} \]

Note: \( a^n \) means the concatenation of \( n \) a’s
A grammar for expressions

N = {E, T, F} where:
   E: expression
   T: term – T = {+, *, (, ), a}
   F: factor

P: the productions:
   1. E -> E + T
   2. E -> T
   3. T -> T * F
   4. T -> F
   5. F -> (E)
   6. F -> a

S = E
Applying the grammar

• You can’t reach a valid conclusion if you don’t have a valid string, but the opposite is not true.

• For example, suppose we want to parse the string \((a * a) + a\) using the grammar we just saw.

• First attempt:
  
  \[ E \rightarrow T \text{ (by rule 2)} \]
  
  \[ \rightarrow F \text{ (by rule 4)} \]

  … and, we’re stuck, because \(F\) can only produce \((E)\) or \(a\); so we reach a dead end, even though the string is valid.
Applying the grammar

• Here’s a parse that works for \((a^*a)+a\):
  
  \[
  E \Rightarrow E + T \text{ (rule 1)} \\
  \Rightarrow T + T \text{ (rule 2)} \\
  \Rightarrow F + T \text{ (rule 4)} \\
  \Rightarrow (E) + T \text{ (rule 5)} \\
  \Rightarrow (T) + T \text{ (rule 2)} \\
  \Rightarrow (T^*F) + T \text{ (rule 3)} \\
  \Rightarrow (T^*a) + T \text{ (rule 6)} \\
  \Rightarrow (F^*a) + F \text{ (rule 4 applied twice)} \\
  \Rightarrow (a^*a) + a \text{ (rule 6 applied twice)}
  \]
Deriving a valid string from a grammar

• Arbitrarily pick a nonterminal on right side of current intermediate string & select rules for substitution until you get a string of terminals

• Automatic translators have more difficult problem:
  – given string of terminals, determine if string is valid, then produce matching object code
  – only way to determine string validity is to derive it from the start string of the grammar – this is called parsing
The parsing problem

• Automatic translators aren’t at liberty to pick rules randomly (as illustrated by the first attempt to translate the preceding expression)

• Parsing algorithm must search for the right sequence of substitutions to derive a proposed string

• Translator must also be able to prove that no derivation exists if proposed string is not valid
Syntax tree

• A parse routine can be represented as a tree
  – start symbol is the root
  – interior nodes are nonterminal symbols
  – leaf nodes are terminal symbols
  – children of an interior node are symbols from right side of production rule substituted for parent node in derivation
Syntax tree for \((a^2+a)+a\)
Grammar for a programming language

• A grammar for a subset of the C++ language is laid out on pages 340-341 of the textbook

• A sampling (suitable for either C++ or Java) is given on the next couple of slides
Rules for declarations

<declaration> -> <type-specifier><declarator-list>;
<type-specifier> -> char | int | double
(remember, this is *subset* of actual language)
<declarator-list> -> <identifier> |
                   <declarator-list> , <identifier>
<identifier> -> <letter> |
              <identifier><letter> |
              <identifier><digit>
<letter> -> a|b|c| ... |z|A|B|...|Z
<digit> -> 0|1|2|3|4|5|6|7|8|9
Rules for control structures

<selection-statement> ->
  if (<expression>) <statement> | 
  if (<expression>) <statement>
  else <statement>

<iteration-statement> ->
  while (<expression>) <statement> | 
  do <statement> while (<expression>) ;
Rules for expressions

\(<\text{expression-statement}\> \rightarrow \ <\text{expression}\> \ ;

\(<\text{expression}\> \rightarrow \ <\text{relational-expression}\>

| \ <\text{identifier}\> = \ <\text{expression}\>

\(<\text{relational-expression}\> \rightarrow \n
\ <\text{additive-expression}\> | \ <\text{relational expression}\> < \ <\text{additive-expression}\> | \ <\text{relational expression}\> > \ <\text{additive-expression}\> | \ <\text{relational expression}\> <= \ <\text{additive-expression}\> | \ <\text{relational expression}\> >= \ <\text{additive-expression}\>

\text{etc.}
Backus-Naur Form (BNF)

• BNF is the standardized form for specification of a programming language by its rules of production
• In BNF, the -> operator is written ::= 
• ALGOL-60 first popularized the form
BNF described in terms of itself (from Wikipedia)

\[ \text{<syntax>} ::= \text{<rule>} \mid \text{<rule>} \text{<syntax>} \]

\[ \text{<rule>} ::= \text{<opt-whitespace}> "<" \text{<rule-name>} ">" \text{<opt-whitespace>} ":=" \text{<opt-whitespace>} \text{<expression>} \text{<line-end>} \]

\[ \text{<opt-whitespace>} ::= "" \mid '"" \]
\[ <!-- "" is empty string, i.e. no whitespace --> \]

\[ \text{<expression>} ::= \text{<list>} \mid \text{<list>} "|" \text{<expression>} \]

\[ \text{<line-end>} ::= \text{<opt-whitespace>} \text{<EOL>} \mid \text{<line-end>} \text{<line-end>} \]

\[ \text{<list>} ::= \text{<term>} \mid \text{<term>} \text{<opt-whitespace>} \text{<list>} \]

\[ \text{<term>} ::= \text{<literal>} \mid "<" \text{<rule-name>} ">" \]

\[ \text{<literal>} ::= "" \text{<text>} "" \mid "" \text{<text>} "" \]
\[ <!-- actually, the original BNF did not use quotes --> \]
Finite State Machines

• Diagram consisting of:
  – nodes, which represent finite states
  – arcs, which connect nodes
  – arcs represent transitions from one state to another

• Can be used to express language syntax
Finite State Machines

• Each FSM has a single start state (with an incoming arrow) and one or more final states, represented by a double circle: ○

• FSM can also represent incorrect syntax, illustrating dead ends – the next slide shows an example
FSM to parse an identifier

A: start state
B: final state
C: dead end; only reachable via incorrect syntax

Start

A

letter
digit

B

letter
digit
digit

C

letter
## State Transition Table

<table>
<thead>
<tr>
<th>Current state</th>
<th>Next state</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Letter</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>
Alternate version: FSM without fail state

![Diagram of an FSM with states A and B, transitions for letter and digit, and a table showing state transitions for letter and digit inputs]
Detecting illegal input with FSM

• Conclude in non-finite state (e.g. state C)
• Be unable to make transition (from start state A in alternate FSM, can go to state B with a character, but not with a digit)
Non-deterministic FSM

- Use if you have to decide between two or more transitions when parsing an input string
- At least one state has more than one possible transition state from itself to another state
- The next slide shows a non-deterministic FSM for parsing a signed integer
Non-deterministic FSM
State transition table for non-deterministic FSM

<table>
<thead>
<tr>
<th>Current state</th>
<th>Next state +</th>
<th>Next state −</th>
<th>Next state Digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>B</td>
<td>B, C</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>B, C</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Empty transitions

- Means transition on empty string
- Used for convenience

FSM at left shows another way to describe an integer; bottom transition doesn’t consume an input character – just indicates that sign (+/-) is optional.
Empty transitions

• FSM with empty transition(s) always non-deterministic
• FSM with empty transition(s) can always be converted to FSM without empty transition(s)
• Examples are shown on the next couple of slides
Eliminating empty transition: example 1

Given transition from X to Y on $\epsilon$ and Y to Z on a, construct transition from X to Z on a

Key point: $\epsilon a = a$
Eliminating empty transition: example 2

(a) The original FSM.

(b) The equivalent FSM without an empty transition.
Removing empty transition, example 3

(a) The original FSM.
(b) The equivalent FSM without an empty transition.
FSM and parsing

• Deterministic FSM always better basis for parsing:
  – can’t make wrong choice with valid string, ending up with dead end
  – so dead end always means invalid string

• Removing empty transitions may produce deterministic FSM from non-deterministic – but not always
Multiple token recognizer

• Token: set of terminal characters that has a distinct meaning as a group
• Token usually corresponds to some non-terminal in a language’s grammar
• Examples:
  – non-terminal: <data-type>
  – terminal: int
Multiple token recognizer

• Common use of FSM in translator: detect tokens in source string
• May be different token types that could appear in a particular position in code – for example, in Pep/8 assembly language, a .WORD can be followed by either a decimal or hexadecimal constant – so the assembler needs FSM that can recognize both
Recognizing hex and decimal values

(a) Separate machines for the h# and d# tokens.

(b) One nondeterministic FSM that recognizes the h# or d# token.
Hex/decimal FSM simplified

(a) Removing the empty transitions.
(b) Removing the inaccessible states.
FSM for parsing Pep/8 identifier or symbol definition