03 - Grammars

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College

Outline

Metalanguages and Subject Languages

- We need a way to make a formal specification of a language.
- Languages which describe other languages are called **metalanguages.** Examples:
  - Set Builder Notation
  - BNF - Backus Naur Form
  - Regular Expressions
  - YACC
- Languages described by a metalanguage are called **subject languages.**

Languages and Grammars

- A **language** L is the set of all texts expressible within that language.
- Languages are typically infinite sets.
- An expressible text in some language L is referred to as a **sentence**.
- The set of rules used to verify membership in a language is called a **grammar**.
- A grammar is sometimes also called a **syntax**.

Grammars

A grammar G is defined as follows:

$$G = \langle T, N, S, P \rangle$$

- *T* A finite set of basic symbols (called an **alphabet**). These symbols are called **terminal symbols** of the language.
- *N* A finite alphabet of **non-terminal symbols**. Members of N label parts of sentences in L.
- *S* $S \in N$ is the **distinguished symbol** or **start symbol**. This is the name of entire sentences.
- *P* A set of productions. Set of rules which transform strings of terminal and non-terminal symbols into a valid sentence.

Example Language: Simplified Email Addresses

Example Sentence:

`robert.lowe@maryvillecollege.edu`

$$
\begin{aligned}
T =& \{\mathrm{a-z}, \mathrm{A-Z}, 0-9, ., \texttt{@}\} \\
N =& \{< \text{email} >, < \text{user} >, < \text{subdomain} >, \\
& < \text{domain} >, < \text{tld} >\} \\
S =& < \text{email} >
\end{aligned}
$$

Email Syntax Productions (*P*)

Productions (*P*)

$$
\begin{aligned}
< \text{email} > &\Rightarrow < \text{user} > \texttt{@} < \text{domain} > \\
< \text{user} > &\Rightarrow \{\text{a} - \text{z}, \text{A} - \text{Z}, 0 - 9, .\} < \text{user} > \\
< \text{domain} > &\Rightarrow \{< \text{subdomain} > . < \text{tld} >, \\
&\qquad < \text{subdomain} > . < \text{domain} >\} \\
< \text{tld} > &\Rightarrow \{\text{com}, \text{org}, \text{edu}\}
\end{aligned}
$$

Synthesis of an Email Sentence

$$
\begin{aligned}
<\text{email}> &\Rightarrow <\text{user}>\ \texttt{@} <\text{domain}> \\
&\Rightarrow \texttt{robert.lowe@} <\text{domain}> \\
&\Rightarrow \texttt{robert.lowe@} <\text{subdomain}> . <\text{tld}> \\
&\Rightarrow \texttt{robert.lowe@maryvillecollege.} <\text{tld}> \\
&\Rightarrow \texttt{robert.lowe@maryvillecollege.edu}
\end{aligned}
$$

Some Properties of Grammars and Their Compilers

- T and N must be disjoint.
- Elements in N must be distinguishable in some way.
- Each production must be distinguishable.
- The work of a compiler with these definitions in mind is:
  1. Verify that every symbol in a program is in T.
  2. Reduce the program string to a string over N.
  3. Find a series of productions in P which produce S.
  4. If no such series of productions is possible, this is not a valid program!
  5. On success, generate code.
- **Discuss:** Does this process have anything to do with closure?

Rules for Productions

- Each production is a pair of strings (p, q).
- p and q contain members of T or N (or both).
- A production rule is a rule where replacing p by q is allowed.
- We often write $p \to q$ or $p ::= q$ for each production.
- The p must contain at least one non-terminal. **Discuss** Why? (HINT: The word "terminal" is not just clever phrasing!)
- The rule of substitution:
  - Suppose $V \Rightarrow W$ where $V, W \in (N \cup T)^*$
  - This holds if we can decompose V and W as:
  - $V = XV'Y$
  - $W = XW'Y$
  - Where there exists production $V' \to W'$.
- Let's do this for the email grammar!

Chomsky Hierarchy

- Let $a \in T$
- Let $A, B \in N$
- Let $\alpha, \beta, \gamma \in (N \cup T)^*$ where $\gamma \neq \lambda$

Type-0 **Recursively Enumerable**
$\alpha A \beta \rightarrow \beta$

Type-1 **Context-Sensitive**
$\alpha A \beta \rightarrow \alpha \gamma \beta$

Type-2 **Context-Free**
$A \rightarrow \alpha$

Type-3 **Regular**
$A \rightarrow a$ and $A \rightarrow aB$

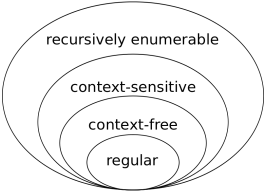

Image Source: `https://en.wikipedia.org/wiki/Chomsky_hierarchy`

Grammars for Programming Languages

- Recursively Enumerable (Type-0) grammars are generally too powerful to describe programming languages. (They would also be too difficult to compile!)
- Most programming languages do contain at least a handful of context-sensitive attributes, though writing true context sensitive grammars is not needed in most cases.
- Context-Free (Type-2) grammars will be our focus. Even in context-sensitive languages, the syntax is often expressed first as a context free grammar with additional constraints applied by the compiler.
- Regular Grammars are too limited to express general programming languages. They are typically useful for searching and general pattern matching.

Backus Naur Form (BNF)

- BNF is a metalanguage which describes context-free grammars.
- Invented by Naur, edited by Backus, first used to describe Algol-60.
- By far, BNF is the most popular metalanguage notation.
- BNF non-terminals are described using meaningful phrases enclosed in angle brackets $\langle\rangle$.
- Replacement arrows are $::=$
- Alternate replacements are separated with $|$.
- Terminal strings are enclosed in single quotes.

Example Grammar

$\langle S \rangle$ ::= <Expression>

$\langle Expression \rangle$ ::= <Term> | <Expression> ‘+’ <Term>

$\langle Term \rangle$ ::= <Factor> | <Term> ‘*’ <Factor>

$\langle Factor \rangle$ ::= <Unit> | (<Expression>)

$\langle Unit \rangle$ ::= ‘0’|‘1’|‘2’|‘3’|‘4’|‘5’|‘6’|‘7’|‘8’|‘9’

The Ledgard Programming Language

- Ledgard is a teaching language.
- Designed by Lee Wittenberg in order to teach students how to write compilers.
- Named in honor of Henry Ledgard, author of *Programming Language Landscapes*.
- Essentially contains “just enough” of the elements of programming languages to explore compiler creation.

Ledgard Syntax

⟨*program*⟩ ::= ‘program’ <decl-list> ‘begin’ <stmt-list> ‘end’ ‘;’

⟨*decl-list*⟩ ::= <declaration> | <decl-list> <declaration>

⟨*declaration*⟩ ::= <identifier-list> ‘:’ <type> ‘;’

⟨*identifier-list*⟩ ::= <identifier> | <identifier-list> ‘,’ <identifier>

⟨*type*⟩ ::= <simple-type> | <array-type>

⟨*simple-type*⟩ ::= ‘integer’ | ‘boolean’

⟨*array-type*⟩ ::= ‘array’ ‘[’ <bounds> ‘]’ ‘of’ <type>

⟨*bounds*⟩ ::= <integer-literal> ‘..’ <integer-literal>

Ledgard Syntax (continued)

⟨*stmt-list*⟩ ::= <statement> | <stmt-list> <statement>

⟨*statement*⟩ ::= <assignment-stmt> | <exchange-stmt> |
<if-stmt> | <loop-stmt> | <input-stmt> | <output-stmt>

⟨*assignment-stmt*⟩ ::= <variable> ':=' <expression> ';'

⟨*exchange-stmt*⟩ ::= <variable> ':=:' <variable> ';'

⟨*if-stmt*⟩ ::= 'if' <expression> 'then' <stmt-list> 'end' 'if' ';'
| 'if' <expression> 'then' <stmt-list> 'else' <stmt-list> 'end' 'if' ';'

⟨*loop-stmt*⟩ ::= 'while' <expression> 'loop' <stmt-list> 'end' 'loop' ';'

Ledgard Syntax (continued)

⟨*input-statement*⟩ ::= ‘input’ <variable-list> ‘;’

⟨*output-statement*⟩ ::= ‘output’ <variable-list> ‘;’

⟨*variable-list*⟩ ::= <variable> | <variable-list> ‘,’ <variable>

⟨*expression*⟩ ::= <operand> | <operand> <operator> <operand>

⟨*operand*⟩ ::= <variable> | <integer-literal> | <boolean-literal> | ‘(’ <expression> ‘)’ | ‘not’ <operand>

⟨*variable*⟩ ::= <variable> | <variable> ‘[’ <expression> ‘]’

Ledgard Syntax (continued)

⟨*boolean-literal*⟩ ::= ‘true’ | ‘false’

⟨*operator*⟩ ::= ‘<’ | ‘<=’ | ‘==’ | ‘<>’ | ‘>=’ | ‘>’ | ‘+’ | ‘-’ | ‘*’ | ‘/’ |
‘and’ | ‘or’

- An integer-literal is just a string of digits 0-9.
- Comments begin with – and continue to the end of the line.