0.1 English language?

- Grammar, syntax - defines rules, the structure of the statement, not the meaning
- Meaning (semantics) - even if structure is correct, meaning may not be direct or correct
- We need to follow the rules, and get the correct meaning

1 Introduction

- Syntax and semantics provide a language's definition
- Syntax: the form or structure of the expressions, statements, and program units
- Semantics: the meaning of the expressions, statements, and program units
- A *sentence (or statement)* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., *, sum, begin)
- A *token* is a category of lexemes (e.g., identifier)
- Programs are string of lexemes rather than characters. E.g., *sum+=2;*

Lexemes and Tokens are closely related:

Lexemes	*Token*
sum	identifier
+	arthmetic operator
=	equal_sign
2	int_literal
;	semicolon

1.1 Recognizers

- A recognition device reads input strings over the alphabet of the language and decides whether the input string belong to the language
- Example: syntax analysis part of a compiler

1.2 Generators

- A device that generates sentences of a language
- One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator

2 BNF and Context-Free Grammers

2.1 Context-Free Grammars

- Developed by Noam Chomsky in the mid-1950s
- Language generators, meant to describe the syntax of natural languages
- Define a class of languages called context-free languages

2.2 Backus-Naur Form

- Inveted by John Backus to describe the syntax of Algol58, later modified by Peter Naur for Algol 60
- BNF (Backus-Naur Form) is equivalent to context-free forms
- In BNF, abstractions are used to represent classes of syntactic structures;they act like syntactic variables, including nonterminal symbols or terminals
- *Terminals* are lexemes or tokens
- A **rule or production** has a left-hand side (LHS) which is a nonterminal, and a right hand side (RHS), which is a string of terminals and/or nonterminals
- Example:

$$< assign > \rightarrow < var > = < expression >$$

- Examples of BNF Rules:
  - $< ident\_list > \rightarrow identifier | identifier, < ident\_list >$
  - $< if\_stmt > \rightarrow if < logic\_expr > then < stmt >$
- A *start symbol* is a special element of the nonterminals of a grammar
- Rules can be recursive

3 Derivation

- A derivation is a repeated application application of rules, starting with the start symbol, repeat till ending with a sentence (all terminal symbols)
- Application of rules:
  - Pick a non-terminal symbol on the right, and replace the non-terminal symbol using a RHS of rule for the non-terminal symbol
  - Example:

$< start\_symbol > \rightarrow < program >$

$< program > \rightarrow \textbf{begin} < stmt\_list > \textbf{end}$

$< stmt\_list > \rightarrow < stmt > \mid < stmt >; < stmt\_list >$

$< stmt > \rightarrow < var > = < expression >$

$< var > \rightarrow A|B|C$

$< expression > \rightarrow < var > + < var > \mid < var > - < var > \mid < var >$

$$
\begin{aligned}
< program > &\rightarrow begin < stmt\_list > end \\
&\rightarrow begin < stmt >; < stmt\_list > end \\
&\rightarrow begin < var > = < expression >; < stmt\_list > end \\
&\rightarrow beginA = < expression >; < stmt\_list > end \\
&\rightarrow beginA = < var > + < var >; < stmt\_list > end \\
&\rightarrow beginA = B + < var >; < stmt\_list > end \\
&\rightarrow beginA = B + C; < stmt\_list > end \\
&\rightarrow beginA = B + C; B = Cend
\end{aligned}
$$

3.0.1 Example

Build a sentence using the following rules:

$$
\begin{aligned}
< assign > &\rightarrow < id > = < expression > \\
< id > &\rightarrow \text{A} \parallel \text{B} \parallel \text{C} \\
< expression > &\rightarrow < id > + < expression > \\
&\quad\ < id > * < expression > \\
&\quad\ (< expression >) \\
&\quad\ < id >
\end{aligned}
$$

3.1 Parse Trees

A hierarchical representation of a derivation. Both a left-most and right-most derivation can be represented by the same parse tree.

3.2 Ambiguity in Grammars

There can sometimes be ambiguities in a language. This can lead to multiple parse trees for the same sentence. Consider the following math expression:

$$A = B + C * D$$

The grammer has no indication of whether the $+$ or the $*$ has any precedence. Our rules may follow standard mathematical notation, giving $*$ precedence. As such, we would have to create more rules to fix precedence.

4 Attribute Grammars

- Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes
- Primary value of AGs
  - Static semantics specification
  - Compiler design (static semantics checking)

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 bf predicates to check for attribute consistency

4.1 Static Semantics

- Nothing to do with meaing: checked when compiling, not executing
- Context-free grammars (CFGs) difficult/cannot describe all of the syntax of programming languages
  - Type compatibility rules
  - All variables must be defined before they are referenced

4.2 Dynamic Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Several needs for a methodology and notation for 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

Three types of Dynamic Semantics:

1. Operational
2. Denontational
3. Axiomatic

4.2.1 Denotational Semantics

$$< bin\_num > \rightarrow `0' \mid `1' \mid \; < bin\_num > `0' \mid \; < bin\_num > `1'$$

- The process of building a denotational specification for a language:
  - Define a mathematical object for each language entity
  - Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects

Example:

$M_{bin}(`0') = 0$

$M_{bin}(`1') = 1$

$M_{bin}(< bin\_num > `0') = 2 * M_{bin}(< bin\_num >)$

$M_{bin}(< bin\_num > `1') = 2 * M_{bin}(< bin\_num >) + 1$