Syntax & Semantics

- Description of programming language
  - For designers
  - For language implementers
  - For programmers
- Syntax
- Semantics

Syntax

- Language is a set of sequences of symbols - sentences
- Syntax defines which sequences define valid sentences
- Symbols called tokens
  - Specific symbols e.g. `=`
  - Sets of symbols e.g. `identifier`
- Defined by lexical rules
- Syntax defined by set of rules (productions)
- Recognizer vs Generator
  - Recognizer determines if a sequence of tokens is part of the language
  - Generator produces sequences of tokens that are part of the language

BNF

- Backus-Naur Form
  - ALGOL 58/60
  - Describes context-free languages
    - Grammar
    - Generator
- Rule (Productions)
  - lhs -> rhs
    - lhs is a name called a non-terminal
    - rhs is a sequence of symbols including non-terminals and tokens (called terminals)
    - Recursion used to allow definition of infinite language
Derivations

- Start symbol
- Substitute for non-terminal on rhs
- Sequence of rule applications is called a derivation
  - Sentential form
  - Leftmost derivation
  - Rightmost derivation

Parse tree

- E.g. assignment statement
- Leaves are terminals
- Internal nodes are non-terminals
- Subtrees are an instance of a rule
- Ambiguity
  - Grammar with two or more parse trees for same sentential form
- Operator precedence
  - Can encode in grammar
- Associativity
  - Left recursive => left associativity
  - Can cause issues for compiler
EXAMPLE 3.2  A Grammar for Simple Assignment Statements

```
<assign> ::= <id> = <expr>
<assign> ::= <assign> , <assign>
<assign> ::= <assign> , <expr>
<assign> ::= <if> , <assign>
<assign> ::= <if> , <expr>
<assign> ::= <id> = <expr>
```

Figure 3.1  An Abstract Syntax Tree for a Simple Assignment Statement:

```
<expr> ::= <id> <op> <id> | <expr> <op> <expr>
<assign> ::= <if> <op> <expr>
```

EXAMPLE 3.3  An Ambiguous Grammar for Simple Assignment Statements

```
<assign> ::= <id> = <expr>
<assign> ::= <assign> , <assign>
<assign> ::= <assign> , <expr>
<assign> ::= <if> , <assign>
<assign> ::= <if> , <expr>
<assign> ::= <id> = <expr>
```
Extended BNF (EBNF)

- Extensions to reduce number of rules
  - Metasymbols
    - `[]`
    - `{ }`
    - `|`
    - `opt`
    - `one of`

Example 3.3: BNF and EBNF versions of an expression grammar

```
BNF:
expr   ::= expr *  term
        ::= term / expr
        ::= term
factor ::= term
        ::= ( expr )
        ::= opt
        ::= one of

EBNF:
expr   ::= expr *  term | term / expr | term
factor ::= term | ( expr ) | opt | one of
```
Semantics

- Meaning of programs
  - Also verifying language rules followed
- Static semantics
  - Things that can be checked at compile time
    - E.g., type checking, defined before use
      - Attribute grammars
- Dynamic semantics
  - Effects of program execution
  - Much harder than syntax
  - No one method dominates

Attribute Grammars

- Augment a grammar
- Attributes
  - Associated with a symbol
  - Have a value
- Semantic functions
  - Associated with a rule
  - Compute attribute values
    - Synthesized attributes
      - Depend only on children nodes (pass up)
    - Inherited attributes
      - Depend on parent and “earlier” siblings (pass down)
- Predicate functions
  - Associated with a rule
  - Specify static semantic rules

E.g. Type Check Assignment

- Grammar
  - Data types: int and real
  - Operands same then that type else real
  - Assignment valid only for same type
- Attributes
  - actual_type
    - Synthesized for <expr>
  - expected_type
    - Intrinsic for <var>
  - expected_type
    - Inherited for <expr>
- Attribute Grammar
  - [i] differentiates between same non-terminals in a rule
Example 3.6: An Algorithm Exercise for Simple Assignment Statement

1. Syntax rule: \( \text{expression} = \text{variable} \)
   - Semantic rule: \( \text{expression} \rightarrow \text{expression} \)
   - Production rule: \( \text{expression} \rightarrow \text{expression} \)

2. Syntax rule: \( \text{expression} = \text{expression} \)
   - Semantic rule: \( \text{expression} \rightarrow \text{expression} \)
   - Production rule: \( \text{expression} \rightarrow \text{expression} \)

3. Syntax rule: \( \text{variable} \)
   - Semantic rule: \( \text{variable} \rightarrow \text{variable} \)
   - Production rule: \( \text{variable} \rightarrow \text{variable} \)

4. Syntax rule: \( \text{variable} \)
   - Semantic rule: \( \text{variable} \rightarrow \text{variable} \)
   - Production rule: \( \text{variable} \rightarrow \text{variable} \)

The task is to formulate a phase-construct tree in the syntax tree and derive the semantics.
Decorating tree
- Computing attribute values for nodes
- Apply semantic functions as values available
- Attributes "flow" up and across tree
Dynamic Semantics

- Describe the meaning of a program execution
- Needed for
  - Programmers to use language
  - Compiler writers to implement compilers
  - Program proving
  - Compiler verification
- No one accepted solution
  - Hard problem
  - Different approaches address specific needs
  - Semantics typically described in natural language

Operational Semantics

- Semantics described by specifying the effect of executing the program
  - Describe the changes to the state of the machine
  - Essentially defining an interpreter for the language
- Level of notation
  - Machine language
  - Intermediate language
- Intermediate language definition
  - Level depends on purpose
  - Its semantics must be obvious and unambiguous
- E.g., C for
  - A simple notation
  - Useful for describing semantics for programmers
Denotational Semantics

- Based on recursive function theory
- For each construct
  - Define a mathematical object which denotes the meaning
  - Define mapping function from construct to object
- Mapping function
  - Domain – parameters – syntactic domain
  - Range – result of mapping – semantic domain
- E.g. binary numbers
  - Syntax
  - Map terminals to mathematical objects (decimal integers)
  - Map rules as function of objects on rhs
  - Semantic function
  - Application
- E.g. decimal numbers
• Representing state
  - Values of program’s variables
  - Set of pairs: <i, v>
    * i - variable name
    * v - value
    - Also undefined
  - \( \text{VARMAP}(i, s) \) value of \( i \) in state \( s \)

• State changes
  - Defined by functions
  - E.g.
    - Expressions
    - Assignment statement
    - Pretest loop
  - Tends to be complex
    - Not as useful for language users
Axiomatic Semantics

- Based on mathematical logic:
  - Describe what can be proven about a program
- Designed for reasoning about programs:
  - Correctness proofs
- Constraints:
  - Rules that define constraints on program variables
  - Use predicate calculus
- Assertions:
  - Logical expressions
  - Precondition
    - Constraints on variables prior to execution of construct
  - Postcondition
    - Constraints on variables after execution of construct
  - E.g. Derive precondition from postcondition
    \[ \text{sum} = 2 \times x + 1 \quad | \quad \text{sum} > 1 \]

- Weakest precondition:
  - Least restrictive precondition that guarantees postcondition
- Proofs:
  - Start with postcondition for program
  - Derive preconditions to beginning
  - Result is the precondition for program
  - If matches the input specification then proved
- Inference rules:
  - Describe truth of an assertion based on values of other assertions
  - Antecedent
  - Consequent
  - Axiom
- Specification:
  - Axiom or inference rule for each construct

E.g.

- Assignment statement:
  - Axiom
  - Rule of consequence
    - Can weaken the postcondition and strengthen the precondition
- Sequence:
  - Inference rule
- Selection:
  - Inference rule
- Pretest loop:
  - Inference rule
  - Loop invariant
  - Requirements
\( P \) = \( Q \) 
\( \text{if } P \text{ and } Q, \text{ then } \) 
\( \text{end if} \)

\( \text{while } P \text{ do } \) 
\( \text{end while} \) 

\( \text{repeat } \) 
\( \text{end repeat} \)