Principles of Programming Languages Midterm Study Guide

Syntax

- **syntax**: a precise description of all its grammatically correct programs
- **lexical syntax**: all the basic symbols of the language
- **concrete syntax**: rules for writing expressions, statements and programs
- **abstract syntax**: internal representation of the program, favoring content over form
- **metalanguage**: a language used to define other languages
- **grammar**: a metalanguage used to define the syntax of a language

Backus-Naur Form (BNF): stylized version of a context-free grammar

Set of

- *productions*: P
- *terminal* symbols: T
- *nonterminal* symbols: N
- *start* symbol: $S \in N$

A *production* has the form $A \to \omega$ where $A \in N$ and $\omega \in (N \cup T)$

Parse Trees
A graphical representation of a derivation.

- each internal node of the tree corresponds to a step in the derivation
- the children of a node represents a right-hand side of a production
- each leaf node represents a symbol of the derived string, reading from left to right

Associativity and Precedence

A grammar is **ambiguous** if one of its strings has two or more different parse trees.

Extended BNF (EBNF)

BNF:

- recursive for iteration
- nonterminals for grouping

EBNF: additional metacharacters:

- for a series of zero of more
- ( ) for a list, must pick one
- [] for an optional list, pick one or none

We can always write an EBNF grammar as a BNF grammar

Identifier: sequence of letters and digits, starting with a letter

Concrete Syntax: based on a parse of its Tokens

Lexer:

- input: characters
- output: tokens
- separate
  - speed: 75% of time for non-optimizing
  - simpler design
  - character sets
  - end of line convention

Parser:

- Based on BNF/EBNF grammar
- Input: tokens
- Output: abstract syntax tree (parse tree)
- Abstract syntax: parse tree with punctuation, many nonterminal discarded

Semantic Analysis

- Check that all identifiers and declared
- Perform type checking
- Insert implied conversion operators (i.e., make them explicit)

Code Optimization

- Evaluate constant expressions at compile-time
- Reorder code to improve cache performance
- Eliminate common subexpressions
- Eliminate unnecessary code

Code Generation

- Output: machine code
- Instruction selection
- Register management
- Peephole optimization

Interpreter

- Replaces last 2 phases of a compiler
- Input:
  - Mixed: intermediate code
  - Pure: stream of ASCII characters
- Mixed interpreters
  - Java, Perl, Python, Haskell, Scheme
- Pure interpreters
  - most Basic, shell commands

Binding: an association between an entity (such as a variable) and a property (such as its value)

- **static**: if the association occurs before run-time
- **dynamic**: if the association occurs at run-time
- The **lifetime** of a variable name refers to the time interval during which memory is allocated

Scope: the collection of statements which can access the name binding

- **static scoping**: a name is bound to a collection of statements according to its position in source program
- same as **lexical scoping**

Symbol Table: a data structure kept by a translator that allows it to keep track of each declared name and its binding

Dynamic Scoping: in dynamic scoping, a name is bound to its most recent declaration based on the program's call history

Overloading: uses the number or type of parameters to distinguish among identical function names or operators

Lifetime: the time interval during which the variable has been allocated a block of memory

type: a collection of values and operations on those values

type error: any error that arises because an operation is attempted on a data type for which it is underfined

- **type system**: provides a basis for detecting type errors
- **statically typed**: the types of all variables are fixed when they are declared at compile time
- **dynamically type**: the type of a