Describing languages

- need a way to describe languages so everyone involved can understand/predict exactly how any program should behave
- “everyone” includes programmers, testers, designers of language tools (compilers, debuggers, etc), doc writers, etc
- natural language (english, french, etc) explanations tend to be wordy/convoluted when trying to explain subtleties, nuances
- examples, pseudo-code, and flow charts rarely capture all side effects/special cases unless you provide a huge collection

Suitable description mechanism

- want a precise, unambiguous means of describing both the syntax rules of the language and the meaning (semantics) the the language constructs
- ideally, this should support automated construction of language processing tools (debuggers, compilers, etc)
- use special grammars for formal definition of a language:
  - regular grammars to describe tokens
  - context free grammars (CFGs) to describe syntax
  - augmented grammars to capture semantics (meaning)

Grammars and language specs

- The formal grammars give the “official” definition of the language, and serve as the basis for tool design
- Not readily human readable, so generally also supply natural language descriptions, pictures, examples, etc, but really these are informal translations of the formal defs
- Language definitions evolve over the years, with refinements of the grammars and informal translations, both to correct mistakes and improve the language
- Care needs to be taken with changes that impact backwards-compability with previous definitions of the language

Languages to describe languages

- different kinds of grammar have different limitations in what they can describe (proofs in CSCI 320):
  - regular grammars to describe the basic tokens of a language (the alphabet, keywords, operators, and symbols)
  - context free grammars to describe the structural syntax (format of loops, function definitions, etc)
  - augmented grammars to describe the semantics, meaning (type checking rules, scoping rules, etc)

Typical compiler actions

- compiler developer writes it to follow the grammar rules
    - read source code and puts in standardized form (e.g. strips out comments, standardizes whitespace)
    - run preprocessor to handle macros, templates, etc
    - use regular grammar rules to turn resulting code into a sequence of tokens (keywords, operators, etc), called tokenizing
    - use CFG rules to build abstract representation of the syntax, build a symbol table to keep track of data associated with each symbol found, use augmented rules to determine semantic meaning
    - apply optimizations and generate resulting machine code

Automated compiler generation

- If we put our grammar rules into a standardized format, we can have a program read the grammar rules and generate the compiler for us! (also called compiler-compilers)
- tokenizers use regular grammars to generate the list of tokens (done by programs like lex, flex)
- yacc (yet another compiler compiler), bison, antlr are examples of programs that read a grammar and produce a compiler (often using tools like lex/flex also)
- resulting compilers generally less efficient than those carefully designed ‘by hand’, but the tool does the work for us