Programming Language Syntax

	Microsyntax	Macrosyntax
Specification	Regular expressions	Context-free grammars - expressed in BNF or EBNF
Algorithm	Lexical analysis/scanning	Parsing - LL, top-down, predictive - LR, bottom-up
Input	Symbol/character stream	Token stream
Output	Token stream	Data structure for code generation
Theoretical foundation	Deterministic finite automaton	Deterministic push-down automaton
Tools	lex, flex	yacc, bison

Microsyntax

- Specified using *regular expressions*
  - a character (in some encoding system; once ASCII, can be Unicode)
  - the empty string (∈ or λ)
  - 2 concatenated regular expressions
  - 2 regular expressions separated by |, denoting a choice between the two
  - a regular expression followed by the *Kleene star* (*), denoting zero or more instances of that regular expression
- Example: numeric literal (*unsigned_number*)

digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

unsigned_integer → *digit digit**

unsigned_number → *unsigned_integer*

((. *unsigned_integer*) | ∈)

((e (+ | – | ∈) *unsigned_integer*) | ∈)

Scanning in Programming Languages

- Regular expressions actually have many other uses beyond programming languages: text search/pattern matching, URL rewriting, network protocols
- In the context of programming languages, regular expressions form the specification component of lexical analysis, or scanning

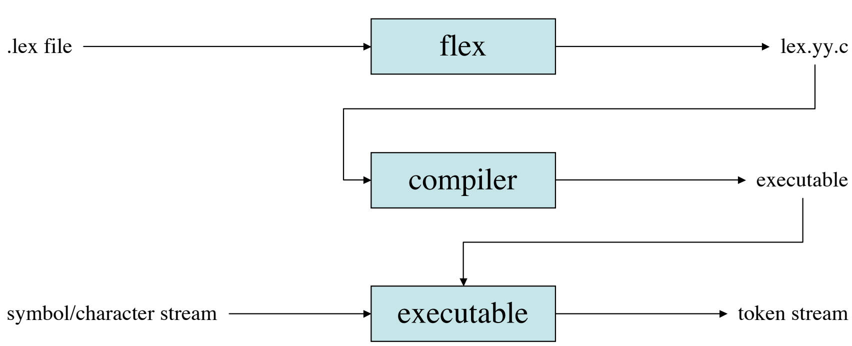
- Beyond that, scanning also…
  - Removes whitespace (spaces, tabs, linefeeds, carriage returns)
  - Removes comments
  - Handle lexical errors (token-level problems; actually quite rare)

- Two styles of scanning
  - Handcoded (actually semi-handcoded — scanners follow the same general pattern)
  - Table-driven (i.e. data-driven)

Scanner Implementation

- Handcoded way = essentially a "writing out" of a finite-state automaton
  - This belongs to Compiler Construction

- Data-driven/table-driven way = use a scanner generator; the best known are *lex* and its newer version, *flex*



Tokens

- Once a token is recognized, two key pieces of information are passed on to the parser:
  - What was recognized (the left side of the regular expression)
  - The exact character sequence that was recognized as this token
  - Special case: reserved words vs. identifiers
    - In Java, *private* is a reserved word, but it is lexically no different from a variable called, say *sarge*
    - To handle this, we "cheat" a little bit by maintaining a separate data structure that lists the reserved words in a language; when an "identifier" is found during lexical analysis, it is looked up against the list of known reserved words, and if there is a match, the token is returned as the reserved word instead of the identifier
  - Examples:
    - "500" is an *integer* with value 500
    - "x" is an *identifier* with value "x"
    - (in C) "return" is a reserved word, so its token is *return*

Macrosyntax

- Specified using *context-free grammars*
  - heuristically: "regular expressions with recursion"
  - standard format: Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF), named after John Backus and Peter Naur
  - historical tidbit: first used to specify Algol-60
  - EBNF is essentially BNF with |, *, and ( ) added
- Example (boldface == terminals == scanner output):

  program → *stmt_list* **$$**
  stmt_list → *stmt stmt_list* | ∈
  stmt → **id :=** *expr* | **read id** | **write** *expr*
  expr → *term term_tail*
  term_tail → *add_op term term_tail* | ∈
  term → *factor factor_tail*
  factor_tail → *mult_op factor factor_tail* | ∈
  factor → **(** *expr* **)** | **id** | **literal**
  add_op → **+** | **–**
  mult_op → ***** | **/**

Parsing in Programming Languages

- Context-free grammars (and parsing in general) actually have many other uses beyond programming languages: speech recognition, document serialization, user interface specification
- In the context of programming languages, context-free grammars form the specification component of syntactic analysis, or parsing

- General parsing of any context-free grammar is $O(n^3)$
- Two context-free grammar categories accommodate $O(n)$ parsing algorithms (i.e. they're practical!)
  - LL (left-to-right, left-most derivation) → top-down or predictive
  - LR (left-to-right, right-most derivation) → bottom-up or shift-reduce

LL(1) and LR(1): 1 token of look-ahead

LL(1)

program → *stmt_list* **$$**
stmt_list → *stmt stmt_list* | ∈
stmt → **id :=** *expr* | **read id** | **write** *expr*
expr → *term term_tail*
term_tail → *add_op term term_tail* | ∈
term → *factor factor_tail*
factor_tail → *mult_op factor factor_tail* | ∈
factor → **(** *expr* **)** | **id** | **literal**
add_op → **+** | **–**
mult_op → ***** | **/**

LR(1)

program → *stmt_list* **$$**
stmt_list → *stmt_list stmt* | ∈
stmt → **id :=** *expr* | **read id** | **write** *expr*
expr → *term* | *expr add_op term*
term → *factor* | *term mult_op factor*
factor → **(** *expr* **)** | **id** | **literal**
add_op → **+** | **–**
mult_op → ***** | **/**

read A
read B
sum := A + B
write sum
write sum / 2
$$

The Pascal *if–then–else*

stmt → **if** *condition then_clause else_clause* | *other_stmt*

then_clause → **then** *stmt*

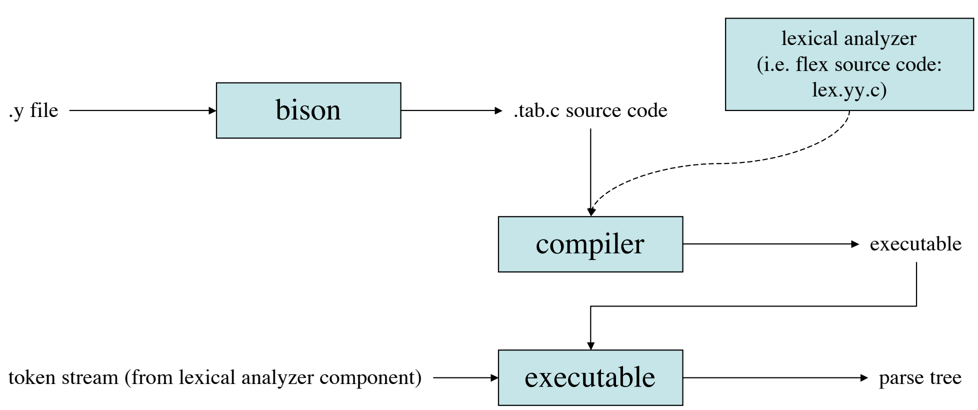
else_clause → **else** *stmt* | ∈

- Ambiguous for "if C_1 then if C_2 then S_1 else S_2"
  - Rewrite the grammar
  - Implement a *disambiguating rule* ("The *else* clause matches the closest unmatched *then*.")
  - Change the syntax!

- Explicit end-markers (*end*, })
- Addition of a separate *elsif* keyword

Parser Implementation

- "The hand way" — LL grammars allow handcoded recursive descent
- Data-driven/table-driven way — more general, and can support bottom-up parsing of LR grammars
  - Doable via parser generators such as *yacc* and *bison*



Implementation Issues

- Look-ahead token(s), or “Real parsers ask for directions”

- The dreaded syntax error, or “Most programmers can’t code the way Mozart composed”
  - Panic mode
  - Phrase-level recovery
    - *first* and *follow* sets
    - historical tidbit: first documented by Wirth for Pascal
  - Context-sensitive lookahead
  - Exception-based recovery
  - Error productions

A Virtual Forest

- Parsing output represents progressively abstract types of data structures, typically best represented a tree (or very similar-looking variant)
- In programming languages, the ultimate goal of parser output is an entity that facilitates code generation and optimization

- Parse trees: a direct mapping from the token stream to the context-free grammar
- Syntax trees: eliminates “helper” tokens and represents the pure syntactic structure of a program
- Abstract syntax trees: static semantics — adds meaning to the symbols of a program, particularly its variables, functions, and other declared entities