

Programming Language Syntax

- Natural language vs. programming language
 - ◆ No ambiguity allowed in programming languages — in form (*syntax*) and meaning (*semantics*)
 - ◆ Thus, programming language specifications use *formal* notations for both syntax and semantics
- Distinction between syntax and semantics: many programming languages have features that *mean* the same (shared semantics) but are *expressed* differently — identifying which is which helps the learning curve

Two Levels of Syntax

	Microsyntax	Macrosyntax
Specification	Regular expressions	Context-free grammars: expressed in BNF or EBNF
Algorithm/Recognition	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

Syntax Specification

- Formalism: set of production rules
- Microsyntax rules: *concatenation*, *alternation* (choice among finite alternatives), “*Kleene closure*”
 - ◆ The set of strings produced by these three rules is a *regular set* or *regular language*
 - ◆ The rules are specified by *regular expressions* — they “generate” the regular language
 - ◆ Strings in the regular language are recognized by *scanners*
- Macrosyntax rules: add *recursion*
 - ◆ The set of strings produced by these rules is a *context-free language* (CFL)
 - ◆ The rules are specified by *context-free grammars* (CFGs) — they “generate” the context-free language
 - ◆ Strings in the context-free language are recognized by *parsers*
- Distinction between a full-fledged *programming* language and a pure *formal* language: Regular and context-free languages are purely formal languages — they are just sets of strings, and do not carry meaning

Microsyntax Specification

- a character (in some encoding system: once ASCII, now Unicode)
- the empty string (ϵ or λ)
- 2 concatenated regular expressions
- 2 regular expressions separated by $|$, denoting a choice between the two regexps
- a regular expression followed by the *Kleene star* ($*$), denoting zero or more instances of that regexp

- Example: numeric literal (*unsigned_number*)
- We use single quotes to identify characters that are expected in the input stream
- Italics indicate regular expressions that are defined elsewhere in the microsyntax

digit \rightarrow '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

unsigned_integer \rightarrow *digit digit**

unsigned_number \rightarrow *unsigned_integer*

(('.' *unsigned_integer*) | ϵ)

(('e' ('+' | '-') | ϵ) *unsigned_integer*) | ϵ)

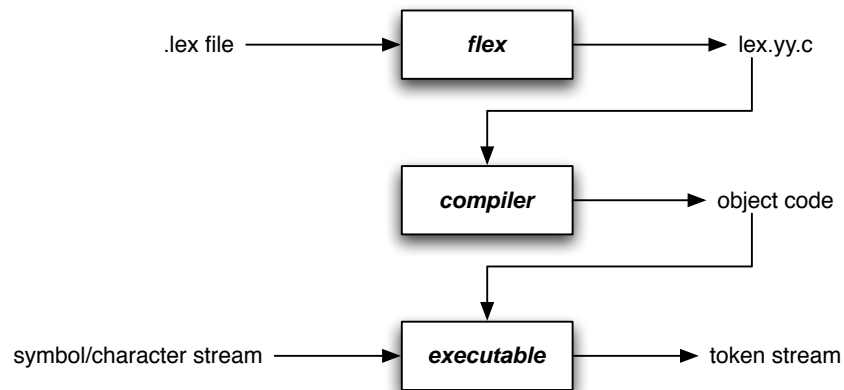
Microsyntax Design Issues

- Reserved word or identifier? Reserved words are their own tokens; identifiers get lex'ed as a single token type with an attached value.
- Case sensitivity in identifiers and keywords?
 - ◆ No: Ada, Common Lisp, Fortran 90, Pascal
 - ◆ Yes: Modula-2/3, C/C++/Java, Perl, ML, JavaScript
- Other identifier issues: what characters to accept, length of identifiers

- White space: mostly *free format* these days — token streams are completely linear, with any amount of white space in between
 - ◆ Early Fortran was not free format: specific character positions had specific functions, with a maximum number of columns (72)
 - ◆ Some languages allow line breaks to separate statements: Haskell, Occam, SR
 - Ditto JavaScript, but really, why? Takes semi-colons just as well, and makes life simpler
- Comments are processed (essentially, removed) during lexical analysis: can be delimited (nested or non-nested?) or to-end-of-line

Microsyntax Implementation

- *Handcoded*: essentially a “writing out” of a finite-state automaton
 - ◇ Details in Compiler Construction
 - ◇ Really *semi*-handcoded: lexical analyzers are a well-known quantity and follow the same overall pattern regardless of the microsyntax — thus, these days, they are almost always...
- *Data-driven/table-driven*: use a scanner generator; the best known are *lex* and its newer version, *flex*



The token stream provides:

1. What was recognized (the left side of the regular expression) — that is, the token itself
2. The exact character sequence that was recognized as this token — the token’s “value” or “spelling”

Reserved Words vs. Identifiers

- In Java, *private* is a reserved word, but it is lexically indistinguishable 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 for that reserved word is returned instead of the identifier token

- Examples:
 - ◆ “500” is an *integer* token with value 500
 - ◆ “x” is an *identifier* token with value “x”
 - ◆ (in C) “return” is a reserved word, so its token is *return* with value as a don’t care
- Interesting footnote: “syntax coloring” features in current programming editors are lexical in nature

Macrosyntax Specification

Heuristically: “regular expressions with recursion”

- Set of *productions* that define *variables* or *nonterminals* in terms of other nonterminals and/or *terminals* (tokens, in the programming languages context)
 - The same nonterminal may be defined by more than one production; terminals never appear on the left side of a production
 - A special nonterminal is “blessed” as the *start symbol*, representing the construct defined by the grammar
-
- Standard formats: Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF), named after John Backus and Peter Naur
 - Historical tidbit: BNF first used to specify Algol-60
 - EBNF is essentially BNF with |, *, and () added
 - ◆ Equivalent expressive power with more convenient notation — compare these productions with their strict BNF versions (bold italic == nonterminal; regular italic == token [terminal]):

operator → *plus* | *minus* | *times* | *divided_by*

identifier_list → *identifier* (*comma identifier*)*

Macrosyntax Design Issues

- We seek a grammar that is *unambiguous* — given a token stream, we can derive one and only one sequence of production rules that results in that stream from the start symbol
- General rule: choose a grammar that reflects the internal structure of the programs it produces
- Particularly important in arithmetic, with its notions of *associativity* and *precedence* of operations

Case Study: Pascal's *if-then-else*

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

then_clause → *then stmt*

else_clause → *else stmt* | ϵ

- Ambiguous for “if C1 then if C2 then S1 else S2”
 - ◇ Rewrite the grammar (e.g. the productions)?
 - ◇ Implement a *disambiguating rule* (“The *else* clause matches the closest unmatched *then*.”)?
 - ◇ ...or change the syntax? (e.g. explicit end markers such as ‘}’ or addition of an *elsif* keyword)

Another Case Study: Expression Parsing

expression \rightarrow *identifier* | *number*
| *negative* **expression** |
| *leftParen* **expression** *rightParen*
| **expression operator expression**

operator \rightarrow *plus* | *minus* | *times* | *divide*

- What derivations in this grammar yield the expression:

slope * x + intercept

- Compare to this grammar, which can parse the same set of token streams:

expression \rightarrow **term** | **expression add_op term**

term \rightarrow **factor** | **term mult_op factor**

factor \rightarrow *identifier* | *number* | *negative* **factor**
| *leftParen* **expression** *rightParen*

add_op \rightarrow *plus* | *minus*

mult_op \rightarrow *times* | *divide*

- Try this for:

slope * x + intercept
x - y - z

Macrosyntax Implementation

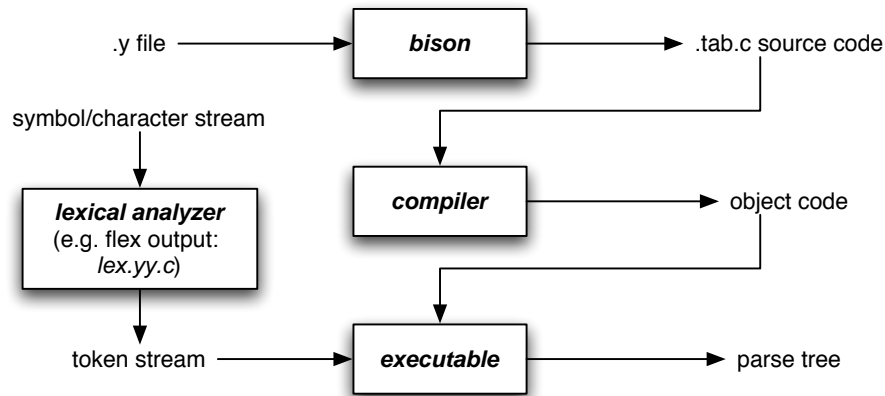
- When creating a program, we take the start symbol of the language's grammar and progressively replace it with some choice of applicable productions until the result consists entirely of terminals
- In reverse (that is, when parsing), we take the stream of terminals (tokens) and determine the sequence of productions that resulted in that stream
- In general, parsing a context-free grammar is $O(n^3)$
- Two CFG categories accommodate $O(n)$ parsing

‘L’ (‘L’ | ‘R’) ‘(n ’)

Grammar Category	Meaning	Algorithm
LL	left-to-right, left-most derivation	top-down, predictive
LR	left-to-right, right-most derivation	bottom-up, shift-reduce

- We add a number in parentheses to indicate the number of *look-ahead tokens* required

- LL grammars allow handcoded implementation: top-down/recursive-descent parsers
- LR grammars require a data-/table-driven implementation, which use a bottom-up/shift-reduce “parser driver”
- ◆ Done by parser generators such as *yacc* and *bison*



- 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

Handling Syntax Errors

- *Panic mode*: Ignore all until we hit a “safe” token
 - *Phrase-level recovery*: Different “safe” sets depending on the current production
 - ◆ Implemented with *first* and *follow* sets
 - *Context-sensitive lookahead*: Refines *follow* sets to be context-specific
 - Historical tidbit: First attempted by Wirth for Pascal
-
- *Exception-based recovery*: Register exception handlers as parser moves through the code; a syntax error “unwinds” to the most recent exception handler
 - *Error productions*: Explicitly define productions in the grammar that represent common or likely syntax errors; when the parser recognizes that production, a very specific error or warning message can be displayed (and corrections even suggested)