classmate
Date 14.8.2019
Page 1

dondi.lmu.build/share/pl/syntax-v02.pdf

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 & 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

	micro syntax	macro syntax
specification	regular expressions	context-free grammers expressed in BNF or EBNF
algorithm / recognition	lexical analysis / scanning	parsing: LL / top-down predictive; LR / bottom-up
input	symbol / character stream	token stream

theoretical boundation	deterministic finite automata	deterministic push-down automata.
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 3 rules is a regular set or a 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).
  - these rules are specified by context-free grammers (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 & context-free langs. are purely formal langs - 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 2 regexps.
- a regular expression followed by the Kleene star (*), denoting 0 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 → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

unsigned_integer → digit digit*

unsigned_number → 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 lexed 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

ORTHOGONALITY OF LANGUAGE
CATCH SYNTAX PROBLEMS - HARD TO DO - AUTOFIX

classmate
Date 14.8.2019
Page 5

- 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 semicolons 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.

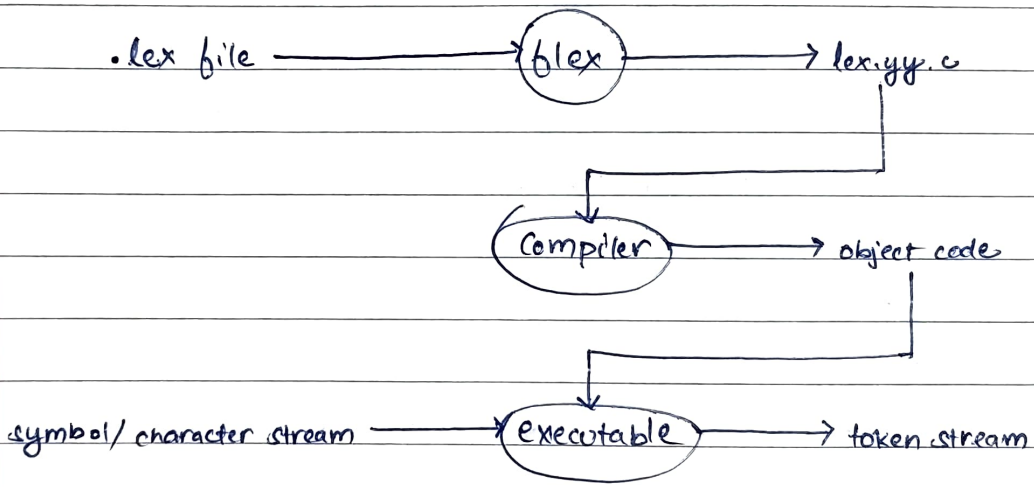
MICROSYNTAX IMPLEMENTATION

- hand coded:
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 ways 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 recursions"

- set of productions that define variables or non-terminals in terms of other non terminals 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 grammer.

- standard formats:
backus-naur form (BNF) or
extended backus-naur form (EBNF),
named after john backus & 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 grammer 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 program 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 grammer (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 '3' or addition of an elsif keyword)

ANOTHER CASE STUDY : EXPRESSION PARSING

expression → identifier | number | negative expression |
left paren expression right paren |
expression operator expression

operator → plus | minus | times | divide

- what derivations in this grammer yield the expression:

  Slope * x + intercept

- compare to this grammer, which can parse the same set of token streams:

expression → term | expression add_op term

term → factor | term mult_op factor

factor → identifier | number | negative factor | leftparen expression rightparen

add_op → plus | minus

mult_op → 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 grammer 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 grammer is $O(n^3)$.

- 2 CFG categories accomodate $O(n)$ parsing

'L' ('L'|'R') 'C' n ')'

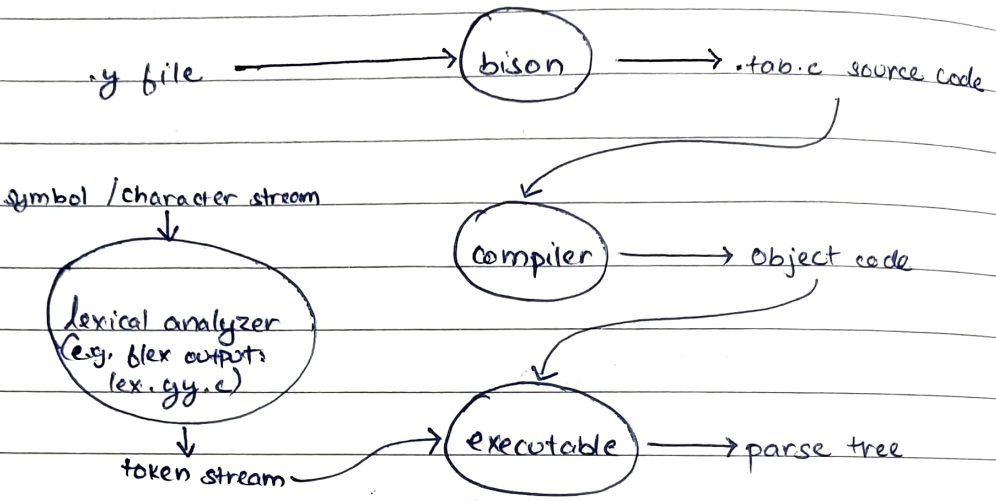
grammer category	meaning	algorithm
LL	left-to-right left-most derivation	top-down, predictive
LR	left-to-right right-most derivation	bottom-up shift-reduce

- we add a nomber in parentheses to indicate the number of look-ahead tokens required.

- LL grammers allow handcoded implementation: top-down / recursive-descent parsers

- LR grammers require a data / table-driven implementation, which use a bottom-up / shift-reduce "parser driver".

  - done by parser generators such as yacc & 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 & optimization.

- parse trees: a direct mapping from the token stream to the context-free grammer.

- 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 upon 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 grammer 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).