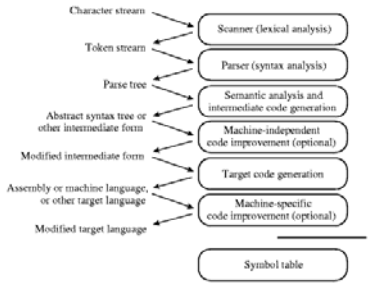
CSE 3302
Programming Languages

Syntax

Chengkai Li
Fall 2007

Phases of Compilation



[Programming Language Pragmatics, by Michael Scott]

Syntax and Semantics

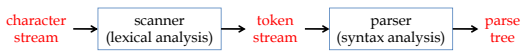
- Defining a programming language:
  - Specifications of syntax
    Syntax – structure (form) of programs (the form a program in the language must take).
  - Specifications of semantics
    Semantics - the meaning of programs
- Precise definition, without ambiguity
  - Given a program, there is only one unique interpretation.

Purpose of Describing Syntax and Semantics

- Purpose
  - For ***language designers***: Convey the design principles of the language
  - For ***language implementers***: Define precisely what to be implemented
  - For ***language programmers***: Describe the language that is to be used
- How to describe?
  - Natural language: ambiguous
  - Formal ways: especially for syntax

Scanning and Parsing

- Lexical Structure: The structure of tokens (words)
  - scanning phase (lexical analysis) : scanner/lexer
  - recognize tokens from characters
- Syntactical Structure: The structure of programs
  - parsing phase (syntax analysis) : parser
  - determines the syntactic structure



Tokens

Tokens (words): Building blocks of programs

- Reserved words (keywords): e.g., `if, while, int, return`
- Literals/constants:
  - numeric literal: 42
  - string literal: `"hello"`
- Special symbols: e.g., `";", "<=", "+"`
- Identifiers: e.g., `x24, monthly_balance, putchar`

Reserved words vs. Predefined identifiers

- Reserved words:
  - cannot be redefined.
    - e.g., `double if;` is illegal.
- Predefined identifiers:
  - have initial meaning
  - allow redefinition (not a good idea in practice)
    - e.g., `String`, `Object`, `System`, `Integer` in Java

Principle of Longest Substring

- The longest possible string of characters is collected into a single token.
- `doif` vs. `do if`.
- An exception: FORTRAN
  - `DO 99 I = 1.10` (the same as `DO99I=1.10`)

White Space

- Principle of longest substring requires that tokens are separated by white space.
- White space (token delimiters):
  - Blanks, newlines, tabs, comments
  - ignored except that they separate tokens
- Free-format language: format has no effect on the program structure
  - Most languages are free format
  - One exception: python

Indentation in Python

```
 def perm(l):                    #error: first line indented
for i in range(len(l)):          #error: not indented
  s = l[:i] + l[i+1:]
    p = perm(l[:i] + l[i+1:])  #error: unexpected indent
    for x in p:
          r.append(l[i:i+1] + x)
      return r                   #error: inconsistent dedent
```

```
def perm(l):
   for i in range(len(l)):
       s = l[:i] + l[i+1:]
       p = perm(l[:i] + l[i+1:])
       for x in p:
           r.append(l[i:i+1] + x)
   return r
```


Regular Expression

- Tokens: defined using grammar or regular expressions.
- Regular expressions: Useful for describing text patterns:
  - e.g., *grep*
- Example:
  `[0-9]+(\.[0-9]+)?`

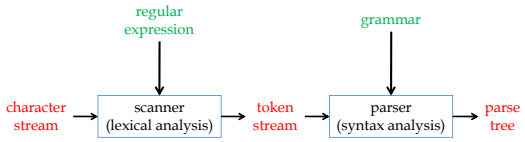
Scanner

regular expression description of the tokens
→ (lex/flex)
scanner of a language

- Example: Figure 4.1 (page 82)

Scanning and Parsing

- Lexical Structure: The structure of tokens (words)
- Syntactical Structure: The structure of programs



Grammar

Example:

(1) *sentence* → *noun-phrase verb-phrase* `.`
(2) *noun-phrase* → *article noun*
(3) *article* → `a` | `the`
(4) *noun* → `girl` | `dog`
(5) *verb-phrase* → *verb noun-phrase*
(6) *verb* → `sees` | `pets`

Figure 4.2 (page 83)

Grammar

- Language: the programs (character streams) allowed
- **Grammar rules** (productions): "produce" the language
  left-hand side, right-hand side
- **nonterminals** (structured names):
  noun-phrase verb-phrase
- **terminals** (tokens): `.` `dog`
- **metasymbols**: → ("consists of") | (choice)
- **start symbol**: the nonterminal that stands for the entire structure (sentence, program).
  - *sentence*
- *E.g., if-statement* → `if` `(` *expression* `)` *statement* `else` *statement*

Grammars Produce Languages

- Language: the set of strings (of terminals) that can be generated from the start symbol by **derivation**:

  sentence ⇒ *noun-phrase verb-phrase* `.` (rule 1)
  ⇒ *article noun verb-phrase* `.` (rule 2)
  ⇒ `the` *noun verb-phrase* `.` (rule 3)
  ⇒ `the girl` *verb-phrase* `.` (rule 4)
  ⇒ `the girl` *verb noun-phrase* `.` (rule 5)
  ⇒ `the girl sees` *noun-phrase* `.` (rule 6)
  ⇒ `the girl sees` *article noun* `.` (rule 2)
  ⇒ `the girl sees a` *noun* `.` (rule 3)
  ⇒ `the girl sees a dog .` (rule 4)

- Define derivation?

Figure 4.3 (page 85)

CFG and BNF

- Context-Free Grammars (CFG)
  - Noam Chomsky, 1950s.
  - Define *context-free languages.*
  - *Four components:*
    - *terminals, nonterminals, one start symbol, productions* (left-hand side: one single nonterminal)
- Backus-Naur Forms (BNF)
  - John Backus/Peter Naur: for describing the syntax of Algol60.
  - BNF: equivalent to CFG
    - Use only metasymbols → | ( )

What does "Context-Free" mean?

- Left-hand side of a production is always one single nonterminal:
  - The nonterminal is replaced by the corresponding right-hand side, no matter where the nonterminal appears. (i.e., there is no *context* in such replacement/derivation.)
- Context-sensitive grammar (context-sensitive languages)
- Why context-free?

Recursion

expr → *expr* + *expr* | *expr* * *expr* | (*expr*) | *number*
number → *number digit* | *digit*
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 4.4 (page 87)

number ⇒ *number digit* ⇒ *number digit digit* ⇒ *digit digit digit*
⇒ 2 *digit digit* ⇒ 23 *digit* ⇒ 234

expr ⇒ *expr* + *expr* ⇒ *expr* + *expr* * *expr*
⇒ … ⇒3 + 4* 5

- How many programs(sentences) are in this language?
- How to derive (1+2) * 3 * (4*5)? How many different derivations? Is the process lengthy and tedious?
- ((1+3)) legal? +(4*5) legal?

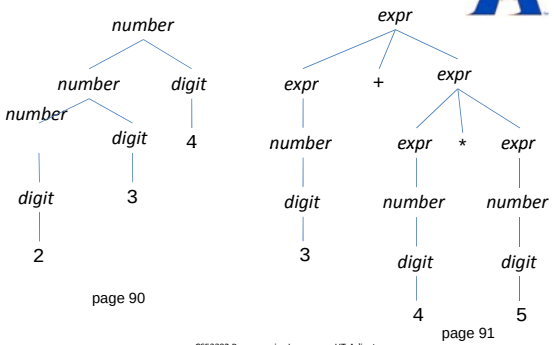
Parse Trees

- There can be many different derivations for capturing the same syntactic structure, and it can become tedious.

Question:
Find another derivation for number ⇒ … ⇒ 234

- Parse trees capture the intrinsic structure in a more intuitive way.

Examples of Parse Trees

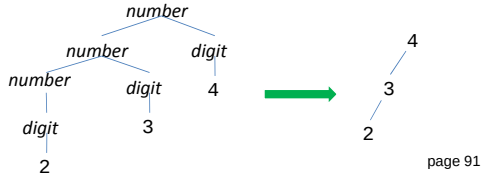


Parse Tree Nodes

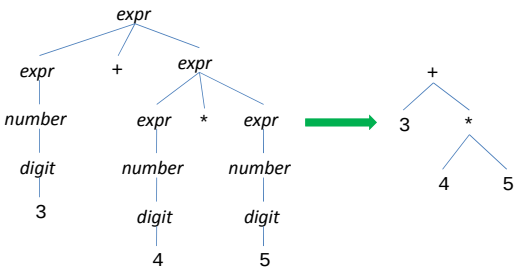
- Root: start symbol
- Leaf nodes: terminals
- Internal nodes: nonterminals
- Children nodes of a node : A replacement (derivation) by the corresponding production rule
  e.g., *expr* → *expr* + *expr*

Abstract Syntax Trees

- Parse Trees: still tedious, all terminals and nonterminals in a derivation are included in the tree.
- Abstract Syntax Tress (Syntax Trees):
  - Remove ``unnecessary'' terminals and nonterminals
  - Still completely determine syntactic structure.



Another Example of Abstract Syntax Tree



Syntax-Directed Semantics (or Semantics-Directed Syntax)

- *sentence → noun-phrase verb-phrase .*

What is meant? (The subject performs some action.)

- *expr → expr + expr*

What is meant? What is the computation executed?