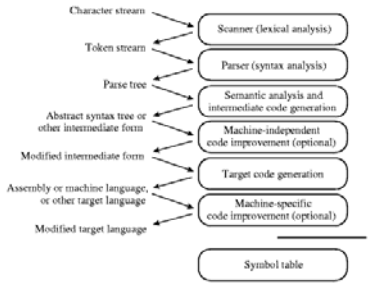
CSE 3302
Programming Languages

Syntax

Chengkai Li, Weimin He
Spring 2008

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

- `doif` vs. `do if;` x12 vs. x 12
- The longest possible string of characters is collected into a single token.
- An exception: FORTRAN
  - `DO 99 I = 1.10` (the same as `DO99I=1.10`)
  - `DO 99 I = 1, 10`

White Space

- Principle of longest substring requires that tokens are separated by white space.
- **White space** (token delimiters):
  - Blanks, newlines, tabs
  - 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

- A form for representing sets of strings
- Description of patterns of characters
- Basic operations:
  - Concatenation
  - Repetition
  - Selection

Regular Expression

Name	**RE**
epsilon	ε
symbol	a
concatenation	AB
selection	A | B
repetition	A*

Shortcuts: $A^+ = AA^*$ A? = A|ε [a-z] = (a|b|...|z)

[a-z][a-z0-9]* (a|b)*aa(a|b)* $[0\text{-}9]^+(\.[0\text{-}9]^+)?$

Tasks of a Scanner

- Recognizes keywords
- Recognizes special characters
- Recognizes identifiers, integers, reals, decimals, strings, etc.
- Ignores white spaces and comments

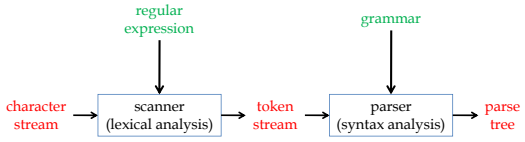
Scanner

regular expression description of the tokens
→ (Lex or JLex)
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)

Context-Free Grammar

- 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)

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?

Backus-Naur Form(BNF)

- A meta language used to describe CFG
- John Backus/Peter Naur: for describing the syntax of Algol60.
- BNF is formal and precise

BNF Example 1

- E → ID
  | NUM
  | E*E
  | E/E
  | E+E
  | E-E
  | (E)

  ID → a | b |…| z

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

BNF Example 2

- S → if E then S else S
  | begin S L
  | print E

  L → end
  | ; S L

  E → NUM = NUM

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

Parse Tree

- Represents the derivation steps from start symbol to the string
- Given the derivations used in the parsing of an input sequence, a parse tree has
  - the start symbol as the root
  - the terminals of the input sequence as leafs
  - for each production $A \rightarrow X_1 X_2 \ldots X_n$ used in a derivation, a node A with children $X_1 X_2 \ldots X_n$

Parse Tree Example 1

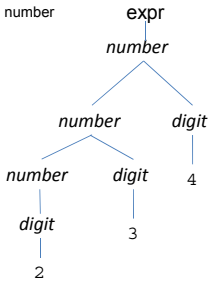
CFG:

expr → expr + expr | expr * expr | (expr) | number

number → number digit |digit

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

Input Sequence: 234



Parse Tree Example 2

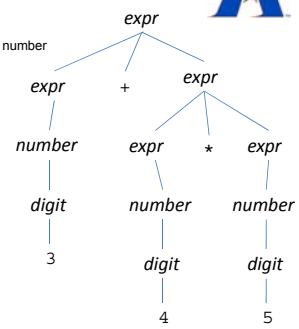
CFG:

expr → expr + expr | expr * expr | (expr) | number

number → number digit |digit

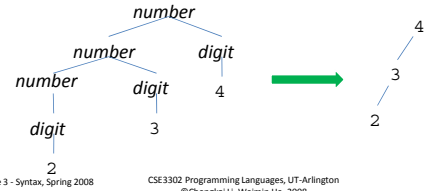
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Input Sequence: 3+4*5

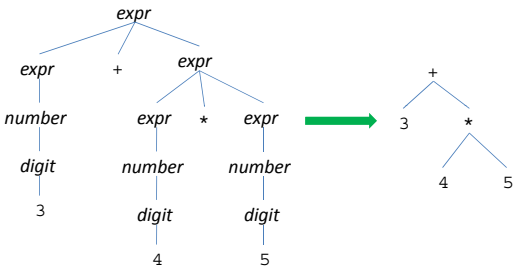


Abstract Syntax Tree

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



AST Example 1



AST Example 2