Phases of Compilation

Character stream → Token stream → Parse tree → Abstract syntax tree or other intermediate form → Modified intermediate form → Assembly or machine language, or other target language → Modified target language → Symbol table

Scanner (lexical analysis) → Parser (syntax analysis) → Semantic analysis and intermediate code generation → Machine-independent code improvement (optional) → Target code generation → Machine-specific code improvement (optional)
Specification of Programming Languages

- PLs require precise definitions (i.e. no ambiguity)
  - Language form (Syntax)
  - Language meaning (Semantics)

- Consequently, PLs are specified using formal notation:
  - Formal syntax
    » Tokens
    » Grammar
  - Formal semantics

Phases of Compilation

Character stream → Token stream → Parse tree → Abstract syntax tree or other intermediate form → Modified intermediate form → Assembly or machine language, or other target language → Modified target language → Symbol table

Scanner (lexical analysis) → Parser (syntax analysis) → Semantic analysis and intermediate code generation → Machine-independent code improvement (optional) → Target code generation → Machine-specific code improvement (optional)
Scanner

- Main task: identify tokens
  - Basic building blocks of programs
  - E.g. keywords, identifiers, numbers, punctuation marks
- Desk calculator language example:
  ```plaintext
  read A
  sum := A + 3.45e-3
  write sum
  write sum / 2
  ```

Formal definition of tokens

- A set of tokens is a set of strings over an alphabet
  - \{read, write, +, -, *, /, :=, 1, 2, ..., 10, ..., 3.45e-3, ...\}
- A set of tokens is a **regular set** that can be defined by comprehension using a **regular expression**
- For every regular set, there is a **deterministic finite automaton** (DFA) that can recognize it
  - i.e. determine whether a string belongs to the set or not
  - Scanners extract tokens from source code in the same way DFAs determine membership
Regular Expressions

- A regular expression (RE) is:
  - A single character
  - The empty string, $\varepsilon$
  - The concatenation of two regular expressions
    - notation: $R_1 R_2$ (i.e. $R_1$ followed by $R_2$)
  - The union of two regular expressions
    - notation: $R_1 \mid R_2$
  - The closure of a regular expression
    - notation: $R^*$
    - $*$ is known as the Kleene star
    - $*$ represents the concatenation of 0 or more strings

Token Definition Example

- Numeric literals in Pascal
  - Definition of the token $\text{unsigned\_number}$

$$\text{digit} \rightarrow 0\mid 1\mid 2\mid 3\mid 4\mid 5\mid 6\mid 7\mid 8\mid 9$$

$$\text{unsigned\_integer} \rightarrow \text{digit\_digit}^*$$

$$\text{unsigned\_number} \rightarrow \text{unsigned\_integer} ( ( . \text{unsigned\_integer} ) \mid \varepsilon )$$

$$= ( ( \varepsilon \mid + \mid - \mid \varepsilon ) \text{unsigned\_integer} ) \mid \varepsilon$$

- Recursion is not allowed!
- Notice the use of parentheses to avoid ambiguity
Scanning

• Pascal scanner

Pseudo-code

we skip any initial white space (spaces, tabs, and newlines)
we read the next character
if it is a `;` we look at the next character
if that is a `*` we have a comment
we skip forward through the terminating `*`
otherwise we return a left parenthesis and reuse the look-ahead
if it is one of the one-character tokens (`[ ] ; = + -` etc.)
we return that token
if it is `.` we look at the next character
if that is a `;` we return `.`
otherwise we return `,` and reuse the look-ahead
if it is a `<` we look at the next character
if that is a `=` we return `<`
otherwise we return `<=` and reuse the look-ahead
etc.
if it is a letter we keep reading letters and digits
and maybe underscores until we can’t anymore.
then we check to see if it is a keyword
if so we return the keyword
otherwise we return an identifier
in either case we reuse the character beyond the end of the token
if it is a digit we keep reading until we find a non-digit
if that is not a `.`, we return an integer and reuse the non-digit
otherwise we keep looking for a real number
if the character after the `.`, is not a digit we return
an integer and reuse the `.`, and the look-ahead
etc.

DFAs

• Scanners are deterministic finite automata

(DFAs)
– With some hacks
Difficulties

- Keywords and variable names
- Look-ahead
  - Pascal’s ranges [1..10]
  - FORTRAN’s example

```
DO 5 I=1,25   => Loop 25 times up to label 5
DO 5 I=1.25   => Assign 1.25 to DO5
```

- NASA’s Mariner 1 (apocryphal?)

- Pragmas: **significant comments**
  - Compiler options

Outline of the Scanner

```
state = start
loop
  case state of
    start:
      save text of current token
      case input char of
        'i', 'I': state = gotdigit
        '[': state = gotcomma
        ',', ']': state = gotcomma
      end case
      case input char of
        '1'..'9', 'A'..'Z':
          state := gotdigit
        '0': state := start
        _else_: state := error
      end case
      _else_
        case input char of
          '0'..'9': state := gotdigit
        _else_: state := error
      end case
      case input char of
        '1'..'9', 'A'..'Z':
          state := gotdigit
        _else_: state := error
      end case
    gotdigit:
      case input char of
        '0'..'9': state := gotdigit
        _else_: state := error
      end case
    gotcomma:
      case input char of
        ',': state := start
        _else_: state := error
      end case
      break loop
    gotdot:
      case input char of
        '0'..'9': state := gotdot
        _else_: state := error
      end case
      case input char of
        '0'..'9': state := gotdot
        _else_: state := error
      end case
      break loop
    gotbrac:
      case input char of
        '{': state := gotbrac
        _else_: state := error
      end case
      break loop
    gotbrace:
      case input char of
        '}': state := start
        _else_: state := error
      end case
      break loop
    gotcolon:
      case input char of
        ':': state := start
        _else_: state := error
      end case
      break loop
    gotperiod:
      case input char of
        '.': state := start
        _else_: state := error
      end case
      break loop
    gotverb:
      case input char of
        'v': state := start
        _else_: state := error
      end case
      break loop
    error:
      if not EOF
        state := start
      end
    end case
  end case
end loop
```
Scanner Generators

- Scanners generators:
  - E.g. lex, flex
  - These programs take a table as their input and return a program (i.e. a scanner) that can extract tokens from a stream of characters

---

Table-driven scanner

Lexical errors

```cpp
state = 1, number of states
actions = record
   action : (move, recognize, error)

numstate : state

tokenfound : token

scan.tab : array [char, state] of actions

keyword.tab : set of record
   keyword : string
   ltoken : token

--- these two tables are created by a scanner generator tool

tok : token

image : string

cur.state : state

cur.char : char

state = start.state

image = null

repeat
  loop
    read cur.char

    case scan.tab[cur.char, cur.state, action
      move:
        cur.state := scan.tab[cur.char, cur.state, newstate
      recognize:
        tok := scan.tab[cur.char, cur.state, token, found

    next inner loop
      error:
        print error message and recover; probably start over

    next inner loop

      append cur.char to image

    until tok not in [white space, comment]

    look image up in keyword.tab and replace tok with appropriate keyword if found

    return (tok, image)
```
Scanners and String Processing

- Scanning is a common task in programming
  - String processing
  - E.g. reading configuration files, processing log files,…

- StringTokenizer and StreamTokenizer in Java
  - [http://java.sun.com/products/jdk/1.2/docs/api/java/util/StringTokenizer.html](http://java.sun.com/products/jdk/1.2/docs/api/java/util/StringTokenizer.html)

- Regular expressions in Python and other scripting languages

Reading Assignment

- Scott’s Chapter 2:
  - Introduction
  - Section 2.1.1
  - Section 2.2.1