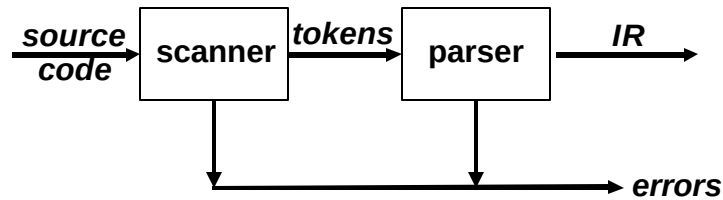


Lecture 5

Lexical Analysis

The scanner is the first component of the front-end of a compiler; parser is the second



The task of the scanner is to take a program written in some programming language as a stream of characters and break it into a stream of tokens. This activity is called *lexical analysis*. A token, however, contains more than just the words extracted from the input. The lexical analyzer partition input string into substrings, called *words*, and classifies them according to their role.

Tokens

A token is a *syntactic category* in a sentence of a language. Consider the sentence:

He wrote the program

of the natural language English. The words in the sentence are: “He”, “wrote”, “the” and “program”. The blanks between words have been ignored. These words are classified as subject, verb, object etc. These are the roles. Similarly, the sentence in a programming language like C:

```
if(b == 0) a = b
```

the words are “if”, “(”, “b”, “==”, “0”, “)”, “a”, “=”, and “b”. The roles are keyword, variable, boolean operator, assignment operator. The pair <role,word> is given the name *token*. Here are some familiar tokens in programming languages:

- Identifiers: x y11 maxsize
- Keywords: if else while for
- Integers: 2 1000 -44 5L
- Floats: 2.0 0.0034 1e5
- Symbols: () + * / { } < > ==
- Strings: “enter x” “error”

Ad-hoc Lexer

The task of writing a scanner is fairly straight forward. We can hand-write code to generate tokens. We do this by partitioning the input string by reading left-to-right, recognizing one token at a time. We will need to look-ahead in order to decide where one token ends and the next token begins. The following C++ code present is template for a `Lexer` class. An object of this class can produce the desired tokens from the input stream.

```
class Lexer
{
    Inputstream s;
    char next; //look ahead
    Lexer(Inputstream _s)
    {
        s = _s;
        next = s.read();
    }

    Token nextToken() {
        if( idChar(next) )return readId();
        if( number(next) )return readNumber();
        if( next == '"' ) return readString();
        ...
        ...
    }
    Token readId() {
        string id = "";
        while(true){
            char c = input.read();
            if(idChar(c) == false)
                return new Token(TID,id);
            id = id + string(c);
        }
    }
    boolean idChar(char c)
    {
        if( isAlpha(c) ) return true;
        if( isDigit(c) ) return true;
        if( c == '_' ) return true;

        return false;
    }
    Token readNumber(){
        string num = "";
        while(true){
```

```

        next = input.read();
        if( !isNumber(next))
            return new Token(TNUM,num);
        num = num+string(next);
    }
}

```

This works ok, however, there are some problem that we need to tackle.

- We do not know what kind of token we are going to read from seeing first character.
- If token begins with “i”, is it an identifier “i” or keyword “if”?
- If token begins with “=”, is it “=” or “==”?

We can extend the `Lexer` class but there are a number of other issues that can make the task of hand-writing a lexer tedious. We need a more principled approach. The most frequently used approach is to use a *lexer generator* that generates efficient tokenizer *automatically*.