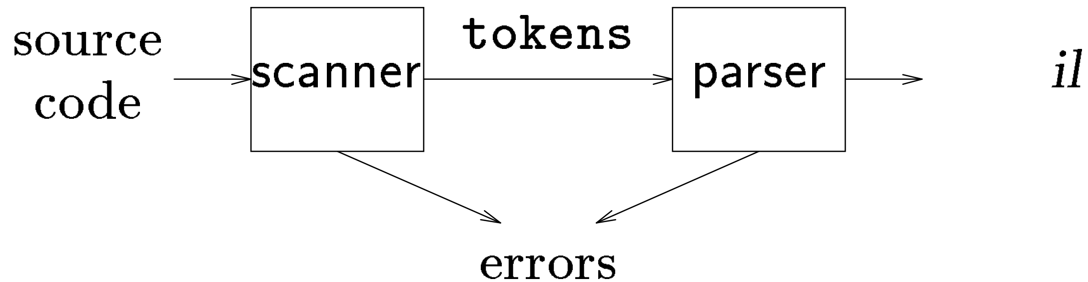
Front end

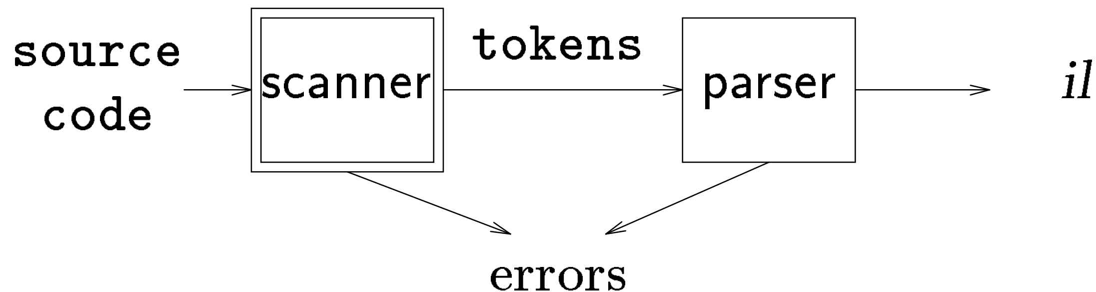


Responsibilities:

- recognize legal procedure
- report errors
- produce *il*
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated

Scanner



A scanner must recognize various parts of the language's syntax.

Input is separated into *tokens* based on lexical analysis.

`x = x + y;`

becomes

<id, `x`> `=` <id, `x`> `+` <id, `y`> `;`

Legal tokens are usually specified by regular expressions (REs).

Regular expressions

Regular expressions represent languages.

Languages are sets of strings.

Operations include *Kleene closure*, *concatenation*, and *union*.

Regular expression	Language
(a)	$\{ \text{"}a\text{"} \}$
$(a) \mid (b)$	$\{ \text{"}a\text{"}, \text{"}b\text{"} \}$
$(a)(b)$	$\{ \text{"}ab\text{"} \}$
$(a)^*$	$\{ \text{""}, \text{"}a\text{"}, \text{"}aa\text{"}, \ldots \}$
$(a)^+$	$\{ \text{"}a\text{"}, \text{"}aa\text{"}, \ldots \}$

We assume *closure*, *concatenation*, *union* as the order of precedence.

$$ab \mid cd^* = (ab) \mid (c(d^*))$$

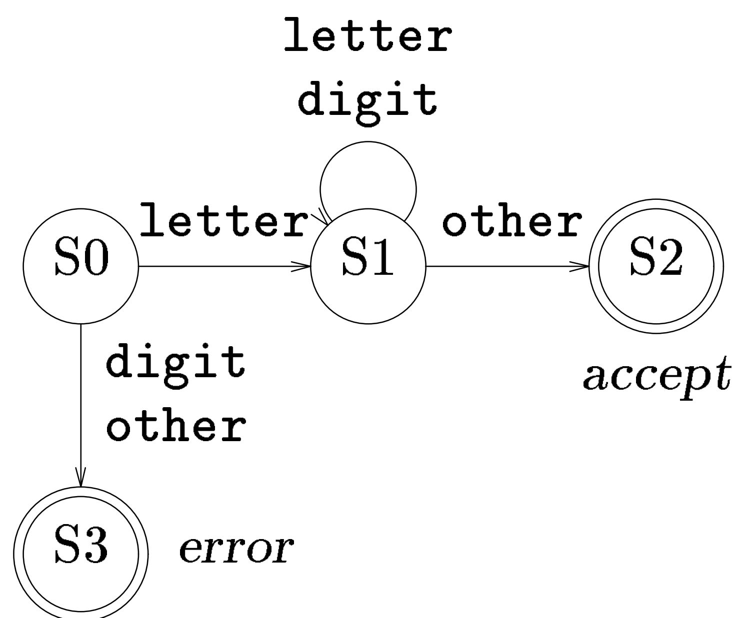
$$= \{ \text{"}ab\text{"}, \text{"}c\text{"}, \text{"}cd\text{"}, \text{"}cdd\text{"}, \ldots \}$$

$$a(bc)^* = \{ \text{"}a\text{"}, \text{"}abc\text{"}, \text{"}abcbc\text{"}, \ldots \}$$

Recognizers

From a regular expression, we can construct a *deterministic finite automaton* (*dfa*).

Recognizer for *identifier*:



identifier

$letter \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z)$

$digit \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)$

$id \rightarrow letter\ (letter \mid digit)^*$

Code for the recognizer

```
char ← next_char();
state ← S0;          /* code for S0 */
done ← false;
token_value ← ""    /* empty string */
while( not done ) {
   class ← char_class[char];
   state ← next_state[class,state];
   switch(state) {
      case S1:     /* building an id */
         token_value ← token_value + char;
         char ← next_char();
         break;
      case S2:     /* accept state */
         token_type = identifier;
         done = true;
         break;
      case S3:     /* error */
         token_type = error;
         done = true;
         break;
   }
}
return token_type;
```

Tables for the recognizer

Two tables control the recognizer.

char_class:

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

next_state:

class	S0	S1	S2	S3
letter	S1	S1	—	—
digit	S3	S1	—	—
other	S3	S2	—	—

To change languages, we can just change tables.

Improved efficiency

Table driven implementation is slow relative to direct code. Each state transition involves:

1. classifying the input character
2. finding the next state
3. an assignment to the state variable
4. a branch
5. a trip through the case statement logic

We can do better by "encoding" the state table in the scanner code.

1. classify the input character
2. test character class locally
3. branch directly to next state

This takes many fewer instructions per cycle.

Faster scanning

```
     token_type ← error;
     char ← next_char();
     class ← char_class[char];
     if (class != letter)
        goto S4;

S1:  token_value ← char;
     char ← next_char();
     class ← char_class[char];
     if (class == other)
        goto S3;

S2:  token_value ← token_value + char;
     char ← next_char();
     class ← char_class[char];
     if (class != other)
        goto S2;

S3:  token_type = identifier;
     return token_type;

S4:  class ← char_class[char];
     return token_type;
```

So what is hard?

Language features that can cause problems:

reserved words

PL/I had no reserved words

```
if then then then = else;
else else = then;
```

significant blanks

FORTRAN and Algol68 ignore blanks

```
do 10 i = 1,25
do 10 i = 1.25
```

string constants

special characters in strings

`newline`, `tab`, `quote`, `comment delimiter`

finite closures

some languages limit identifier lengths

adds states to count length

FORTRAN 66 $\rightarrow$ 6 characters

These problems can be swept under the rug (avoided) by intelligent language design.

Lexical errors

What is a lexical error?

- 1234G6
- illegal character

What should the scanner do?

- report the error
- try to correct it?

Error correction techniques

- minimum distance corrections
- hard token recovery
- skip until match

How bad can it get?

```
1              INTEGERFUNCTIONA
2              PARAMETER(A=6,B=2)
3              IMPLICIT CHARACTER*(A-B)(A-B)
4              INTEGER FORMAT(10),IF(10),DO9E1
5       100    FORMAT(4H)=(3)
6       200    FORMAT(4 )=(3)
7              DO9E1=1
8              DO9E1=1,2
9                 IF(X)=1
10                IF(X)H=1
11                IF(X)300,200
12      300       CONTINUE
13             END
        C      this is a comment
             $ FILE(1)
        14     END
```

Example due to Dr. F.K. Zadeck of IBM Corporation

Automatic construction

Scanner generators automatically construct code from regular expression-like descriptions.

- construct a *dfa*
- use state minimization techniques
- emit code for the scanner
  (table driven or direct code)

A key issue in automation is an interface to the parser.

`lex` is a scanner generator supplied with UNIX.

- emits C code for scanner
- provides macro definitions for each token
  (used in the parser)

`flex` is a modern version of `lex`.