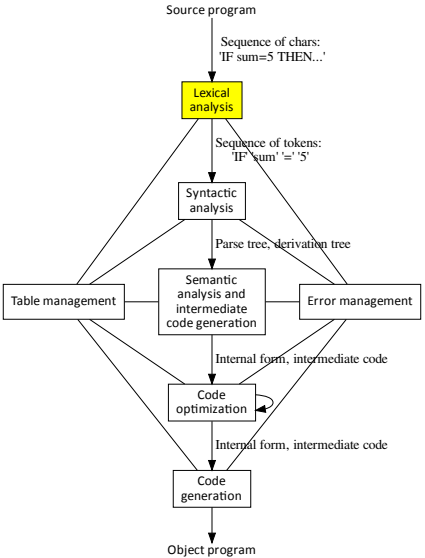
TDDB44/TDDD55 Lecture 4a: Lexical Analysis Scanners

Peter Fritzson and Martin Sjölund

Department of Computer and Information Science
Linköping University

2019-11-21

Lexical Analysis in the Compiler

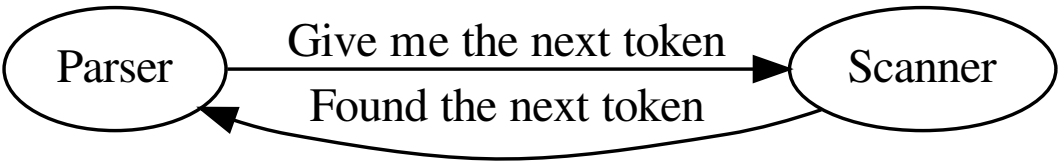


Lexical Analysis, Scanners

- Function
  1. Read the input stream (sequence of characters), group the characters into primitives (tokens). Returns token as `(type, value)`.
  2. Throw out certain sequences of characters (blanks, comments, etc.).
  3. Build the symbol table, string table, constant table, etc.
  4. Generate error messages.
  5. Convert, for example, string $\longrightarrow$ integer.
- Tokens are described using regular expressions
  - See Lecture 3 on Formal Languages to refresh your knowledge of regular expressions, DFAs, NFAs.

Construction of a Scanner

- Tools: state automata and transition diagrams.
- Regular expressions enable the automatic construction of scanners.
- Scanner generate (e.g. Lex, flex):
  - In: Regular expressions
  - Out: Scanner (source code corresponding to a scanner)
- Environment:



How is a Scanner Programmed?

- Describe tokens with regular expressions.
- Draw transition diagrams.
- Code the diagram as table/program.

Example Scanner

```
keyword : 'BEGIN' | 'END' ;
id : letter (letter | digit)* ;
integer : digit+ ;
op : '+' | '-' | '*' | '/' | '//' | '↑' | '=' | ':=' ;
```

There may be blank characters (whitespace) between tokens. This is handled in different ways in different scanner generators:

ANTLR:
```
whitespace : [ \t\n\r]+ {$channel = HIDDEN;} ;
```

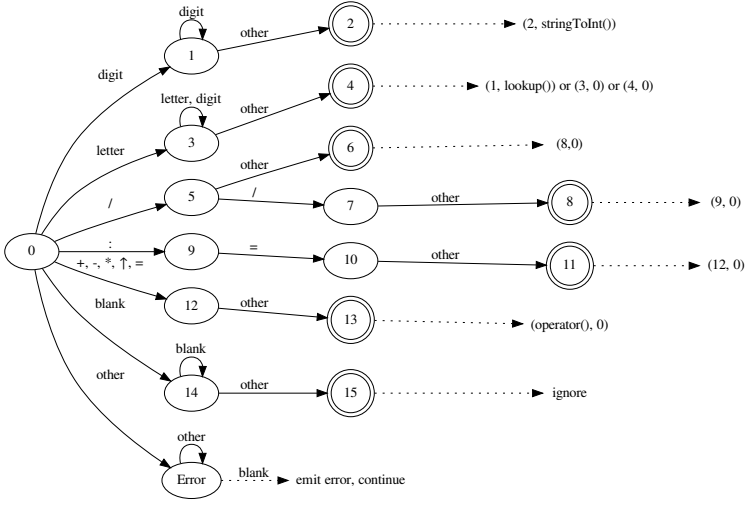
flex:
```
[ \t\n\r]+ ;
```

Scanner Representation of Tokens – Tuples

Input	**(tag, value)**
undef	(0, 0)
id	(1, table-pointer)
integer	(2, value)
`BEGIN`	(3, 0)
`END`	(4, 0)
+	(5, 0)
−	(6, 0)
*	(7, 0)
/	(8, 0)
//	(9, 0)
↑	(10, 0)
=	(11, 0)
:=	(12, 0)

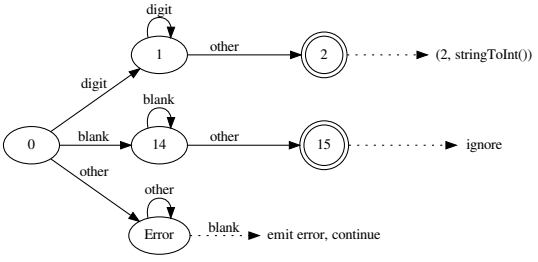
1. Draw the Transition Diagram

- `stringToInt()` converts text to integers.
- `lookup()` returns index to symbol table.
- `BEGIN` and `END` are dealt with by putting them in the symbol table from the beginning. When they are found, return 3 or 4 instead of 1. The benefit of this is a smaller state machine; in many real world examples `BEGIN`, `END` and id would have many states to differentiate between them directly.
- `operator()` converts the matched text into the number of the corresponding tag.
- Automatic transition to state 0 after each recognized token (the input continues with the character read before the recognized token; for example `"12abc"` would return `(2,12)` and continue with `"abc"` as the remainder and not `"bc"`).



2. Translating the Transition Diagram into a Table

Example: Transition diagram and table for only the integer part of the larger diagram.

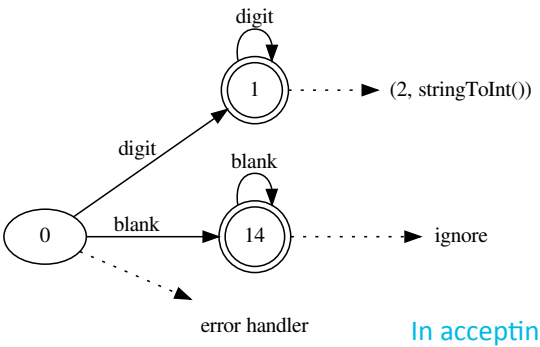


state	digit	blank	other	Action
0	1	14	E	
1	1	2	2	
2	-	-	-	Accept
14	15	14	15	
15				Accept
E				Error

This corresponds to for example Fig. 3.16 in the book.

2. Translating the Transition Diagram into a Table (Compact form)

Example: Transition diagram and table for only the integer part of the larger diagram.



state	digit	blank	other
0	1	14	E
1	1	A	A
14	A	14	A

This is a more compact representation of the transition diagram.
When the next input does not exist in the diagram:

In accepting state: Run the corresponding action

In other state: Run the error handler

3. Interpreting the Table

state	digit	blank	other
0	1	14	-99
1	1	-1	-1
14	-14	14	-14

```
Token scan(FILE *fin) {
  Token t = new Token(); int state = 0; string text = "";
  while (1) {
    char ch = fgetc(fin); int oldstate = state;
    state = table[state][ch]; // transition
    if (has_action(state)) { // Negative numbers are actions
      ungetc(ch, fin); /* Unused value; for next token */
      switch (abs(state)) {
      case 1:
        t->tag = 2; /* integer token */
        t->ival = stringToInt(text);
        return t;
      case 14:
        text = ""; state = 0;
      case 99: // ERROR
        return error_handler(oldstate, ch, t, /* ... */);
      default:
        abort();
      }
    } else {
      text += ch;
    }
  }
```

3. Interpreting the Table

```
// global data structures:
int table [ Nstates ][ Nchars ]
                    = {
  /* ... */
};

typedef struct {
     int tokentype;
     union {
        int ival;
        float fval;
        double dval;
        /* ... */
        symboltable *stptr;
     } tokenval;
} *Token;
```

```
Token scan(FILE *fin) {
  Token t = new Token(); int state = 0; string text = "";
  while (1) {
    char ch = fgetc(fin); int oldstate = state;
    state = table[state][ch]; // transition
    if (has_action(state)) { // Negative numbers are actions
      ungetc(ch, fin); /* Unused value; for next token */
      switch (abs(state)) {
      case 1:
        t->tag = 2; /* integer token */
        t->ival = stringToInt(text);
        return t;
      case 14:
        text = ""; state = 0;
      case 99: // ERROR
        return error_handler(oldstate, ch, t, /* ... */);
      default:
        abort();
      }
    } else {
      text += ch;
    }
  }
```

4. Other Representations of the Table

state	digit	blank	other
0	1	14	-99
1	1	-1	-1
14	-14	14	-14

Using direct goto jumps:

```
state0:
  ch = fgetc(fin);
  if ch >= '0' && ch <= '9' goto state1;
  if ch == " " goto state14;
  goto stateE; /* in other cases */

state1:
  /* ... */
```

Using switch statements:

```
switch (state)  {
  case 0:
    switch (ch) {
      case '0': state = 1; break;
      ...
      case '9': state = 1; break;
      case ' ': state = 14; break;
      default:  state = E; }
      break;
  case 1:      /* ... */
```

5. Direct Coding of Diagrams (not via a table) – Data Structures and Functions

Variables:

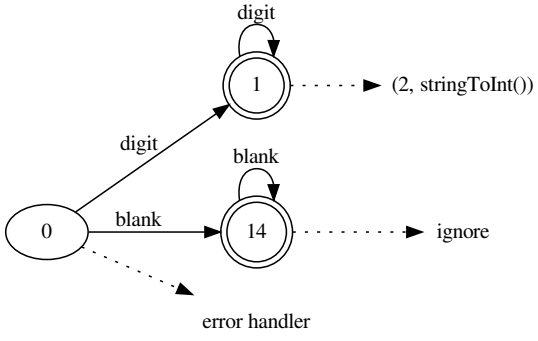
- `t->tokentype` = current symbol class (tag)
- `value`
- `ch` = current character
- `chtyp` = vector for 1-character tokens
- `symtab` = symbol table

Functions:

- `fgetc(fin)`
- `ungetc(ch, fin)`
- `skip_blanks()`
- `is_letter(ch)`
- `symtab_lookup(id)`
- `isalpha(ch)`
- `isdigit(ch)`

Initialize:

- `chtyp` according to the previous description
- Symbol table with reserved words



5. Scanner Fragment with Direct Coding Continued

```
Token getNextToken( FILE *fin )
{
  char ch = fgetc(fin);
  char idstr[BUFSIZE]; // lexeme buffer for identifiers
  t = new_Token();
  while (is_blank(ch)) {
    ch = fgetc(fin); // eat whitespace
  }
  if (isalpha(ch)) {   // identifier:
    while (isalpha(ch) || isdigit(ch) ) {
      append(ch, idstr);
      ch = fgetc(fin);
    }
    ungetc(ch, fin);
    t->tokenval.stptr = symtab_lookup(idstr);
    return t;
  }
  /* ... */
  /* ... */
  else if (is_digit(ch))  { // int-constant:
    int ivalue = ch - '0';
    ch = fgetc(fin);
    while (isdigit(ch)) {
      ivalue *= 10;
      ivalue += ch - '0';
      ch = fgetc(fin);
    }
    ungetc(ch, fin);
    t->tokenval.ival = ivalue;
  }
  /* ... */
  else {
    // others  (single-char. symbols):
    t->tokentype = chtyp[ch];
    if (t->tokentype == 0) {
      error(/* ... */);
    }
  }
  return t;
}
```

Scanner Lookahead Problems

- **Lookahead** is sometimes needed to determine symbol type.
- Example in FORTRAN
  - `DO 10 I = 1.25` is an assignment, but
  - `DO 10 I = 1,25` is a for-statement.

  It is `.` or `,` which determines whether the scanner returns `DO10I` or `DO`.
- Another example in Pascal:
  - Two character lookahead needed for `715..816`

www.liu.se