南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 2

Yepang Liu

liuyp1@sustech.edu.cn

Agenda

- SPL lexical specification
- Recognizing tokens using transition diagram
- Lab task: unique valid identifiers
- Survey of language features

Valid tokens in SPL

- https://github.com/sqlab-sustech/CS323-2024F/blob/main/spl-spec/token.txt

  INT → integer in 32-bits (decimal or hexadecimal like `0xdeadbeef`)

  FLOAT → floating point number (only dot-form)

  CHAR → single ASCII character (printable or hex-form like `'\xa0'`)

  TYPE → int | float | char

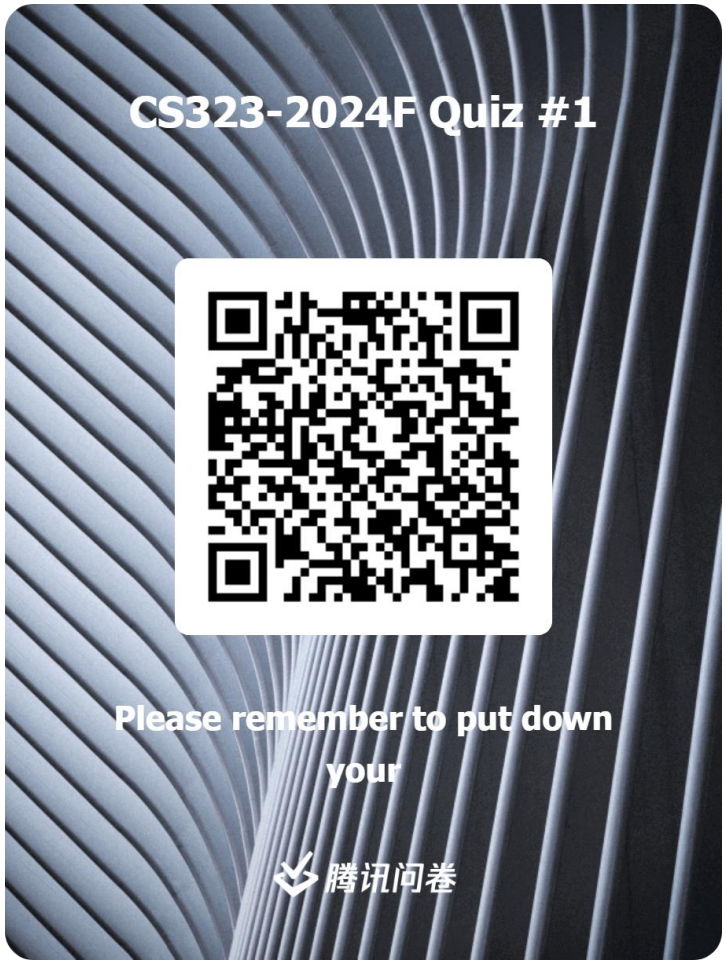
  ID → identifiers consist of 3 types of characters: the underscore (_), digits (0-9), and letters (A-Z and a-z). A valid identifier cannot start with digit.

Valid tokens in SPL

- https://github.com/sqlab-sustech/CS323-2024F/blob/main/spl-spec/token.txt

STRUCT → struct	IF → if	ELSE → else	
WHILE → while	RETURN → return	DOT → .	
SEMI → ;	COMMA → ,	ASSIGN → =	LT → <
LE → <=	GT → >	GE → >=	NE → !=
EQ → ==	PLUS → +	MINUS → -	MUL → *
DIV → /	AND → &&	OR → ||	NOT → !
LP → (	RP →)	LB → [	RB →]
LC → {	RC → }		

Let's Test Your Understanding ☺



https://wj.qq.com/s2/15339168/78d4/

test_1_r02.spl

```
int global;
struct my_struct
{
  int code;
  char data;
};
int test_1_r02()
{
  struct my_struct obj;
  obj.code = global;
  global = global + 1;
}
```

✓ **No lexical errors.**

test_1_r03.spl

```
int test_1_r03()
{
        int i = 0, j = 1;
   float i = $;
   if(i < 9.0){
     return 1
   }
   return @;
}
```

test_1_r06.spl

```
int test_1_r06()
{
    int right_id_1, _right_id_2;
    float 3_wrong_id;
}
```

test_1_r07.spl

```
int test_1_r07()
{
  int a, b, c;
  a = 1;
  b = a && 2;
  c = b || 3;
  a = c | 4;
  b = a & 5;
}
```

test_1_r09.spl

```
int test_1_r09()
{
  int m = 1;
  float n = 2.2;
  int x[5], y[10];
  x[m] = 7;
  y[n] = 8;
  return x[y];
}
```

✓ **No lexical errors.**

test_1_r11.spl

```
int test_1_r11()
{
   int _wrong_hex_int_1 = 0x77G;
   int _wrong_hex_int_2 = 0xCS;
   int _right_hex_int = 0xdeadbeef;
   char _right_hex_char = '\xa0';
   char _wrong_hex_char_1 = '\x6u';
   char _wrong_hex_char_2 = '\x910';
}
```

Agenda

- SPL lexical specification
- Recognizing tokens using transition diagram
- Lab task: unique valid identifiers
- Survey of language features

Recognition of Tokens

- Lexical analyzer examines the input string and finds a prefix that matches one of the tokens
- The first thing when building a lexical analyzer is to define the patterns of tokens using regular definitions
- A special token: `ws` → `(blank | tab | newline)`$^+$
  - When the lexical analyzer recognizes a whitespace token, it does not return it to the parser, but restart from the next character

Example: Patterns and Tokens

digit	→	[0-9]
digits	→	*digit*$^+$
number	→	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	→	[A-Za-z]
id	→	*letter* (*letter* | *digit*)*
if	→	if
then	→	then
else	→	else
relop	→	< | > | <= | >= | = | <>

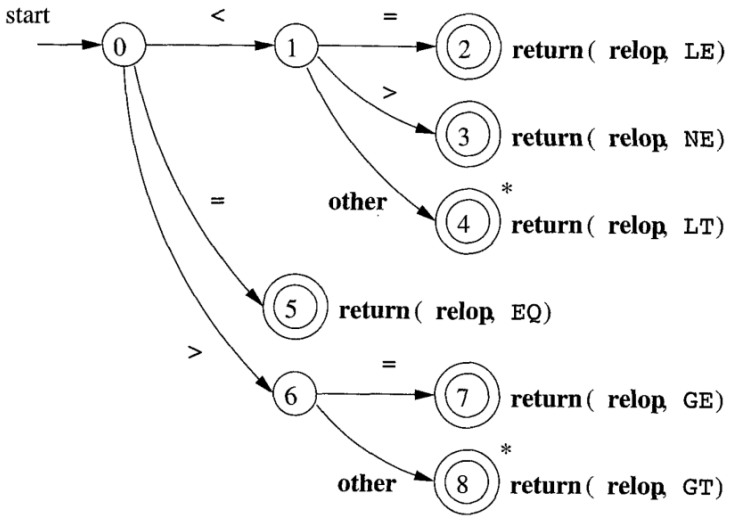
Patterns for tokens

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	if	–
then	then	–
else	else	–
Any *id*	id	Pointer to table entry
Any *number*	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Lexemes, tokens, and attribute values

Transition Diagrams (状态转换图)

- An important step in constructing a lexical analyzer is to convert patterns into "transition diagrams"
- Transition diagrams have a collection of nodes, called *states* (状态) and *edges* (边) directed from one node to another

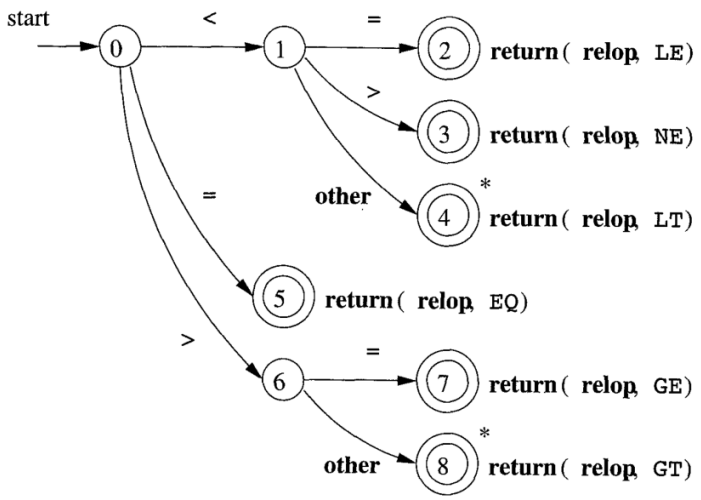


LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

The transition diagram in the left recognizes **relop** tokens

States

- Represent conditions that could occur during the process of scanning (i.e., what characters we have seen)
- The *start state* (开始状态), or *initial state*, is indicated by an edge labeled "start", which enters from nowhere
- Certain states are said to be *accepting* (接受状态), or *final*, indicating that a lexeme has been found

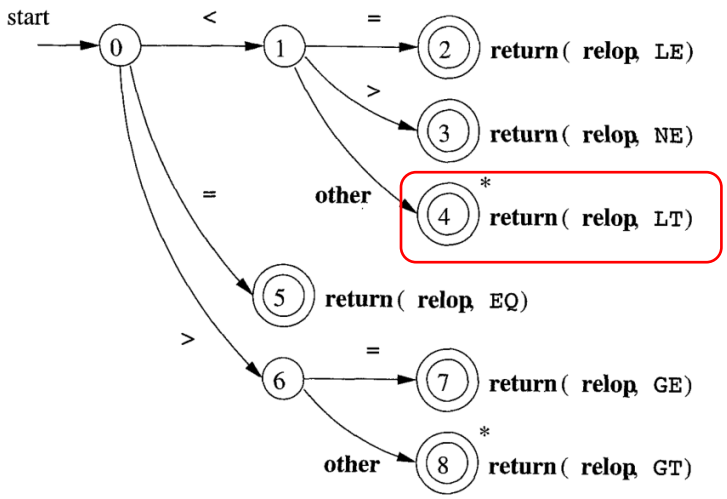


States 2-8 are accepting. They return a pair (token name, attribute value).

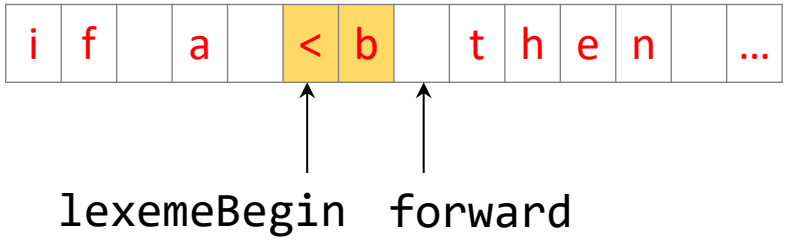
By convention, we indicate accepting states by double circles

The Retract Action

- At certain accepting states, the found lexeme may not contain all characters that we have seen from the start state (such states are annotated with *)
- When entering * states, it is necessary to retract (回退) the `forward` pointer, which points to the next char in the input string



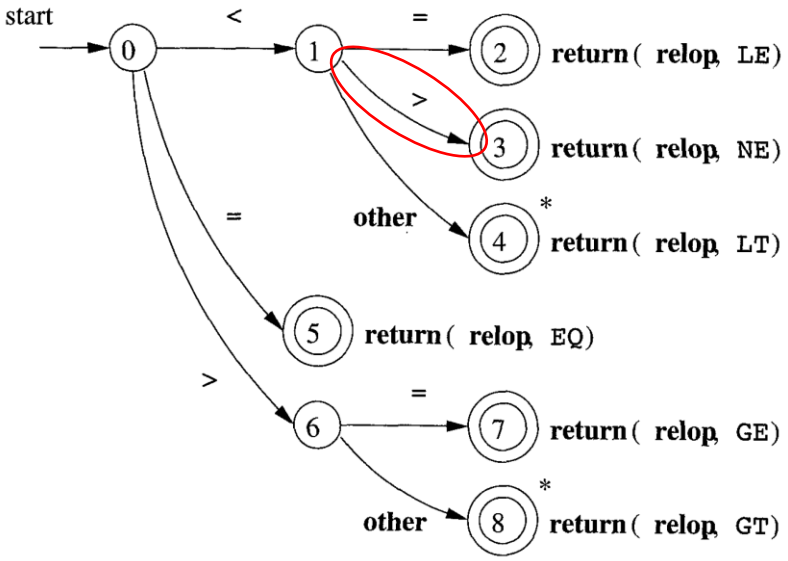
- The found lexeme: <
- The characters we've seen: <b

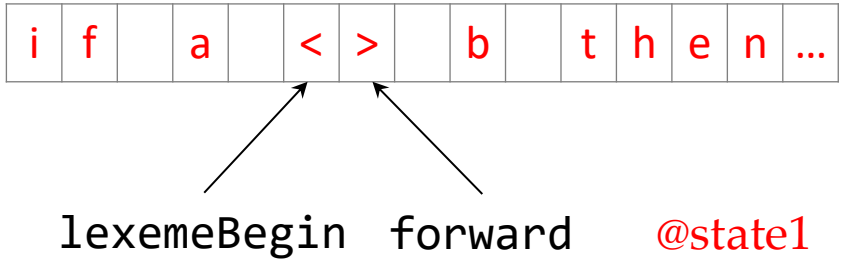


We should retract `forward` one step back

Edges

- *Edges* are directed from one state to another
- Each edge is labeled by a symbol or set of symbols

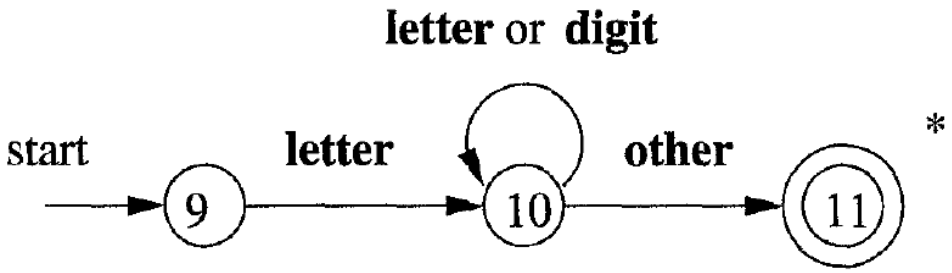




In the above case, we should follow the circled edge to enter state 3 and advance the `forward` pointer

Recognition of Reserved Words and Identifiers (保留字和标识符的识别)

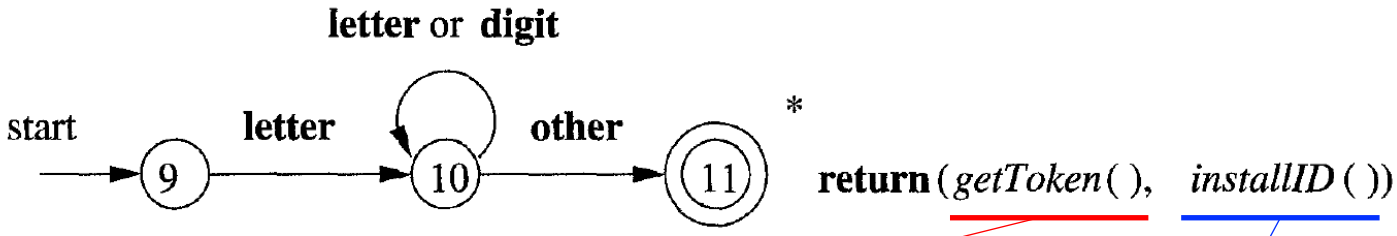
- In many languages, **reserved words** or **keywords** (e.g., `then`) also match the pattern of identifiers
- **Problem:** the transition diagram that searches for identifiers can also recognize reserved words



Is `then` an identifier?

Handling Reserved Words

- **Strategy 1:** Preinstall the reserved words in the symbol table. Put a field in the symbol-table entries to indicate that these strings are not ordinary identifiers (预先存表方案)

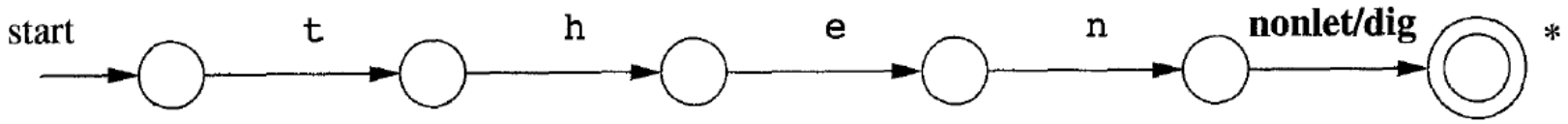


Examine the symbol table and return the token name (identifier or reserved word)

Place a recognized identifier/keyword in the symbol table if it is not already there and return a pointer to the entry

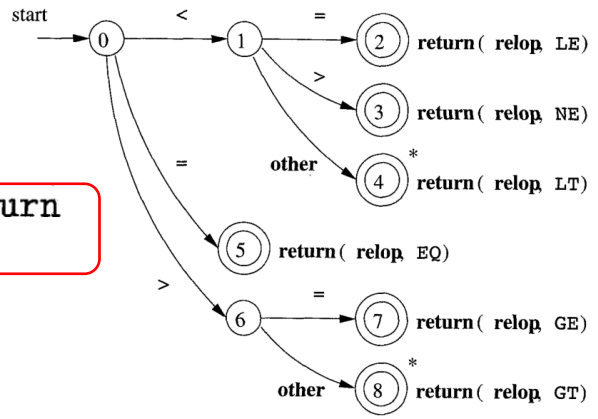
Handling Reserved Words

- **Strategy 2:** Create a separate transition diagram with **a high priority** for each keyword (多状态转移图方案)



Building a Lexical Analyzer from Transition Diagrams

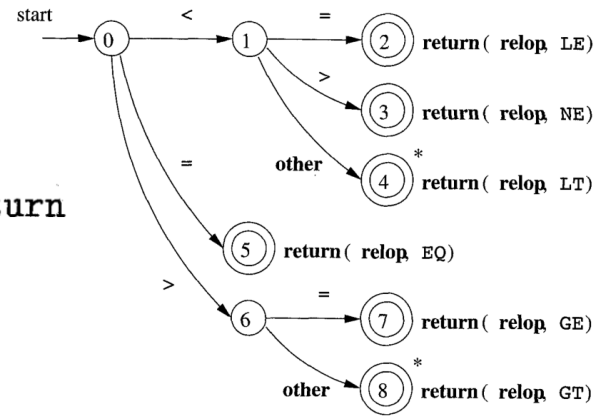
```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```



Sketch implementation of `relop` transition diagram

Building a Lexical Analyzer from Transition Diagrams

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

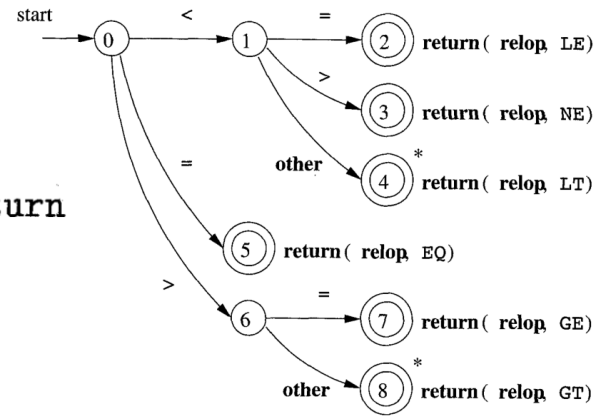


Use a variable `state` to record the current state

Sketch implementation of `relop` transition diagram

Building a Lexical Analyzer from Transition Diagrams

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

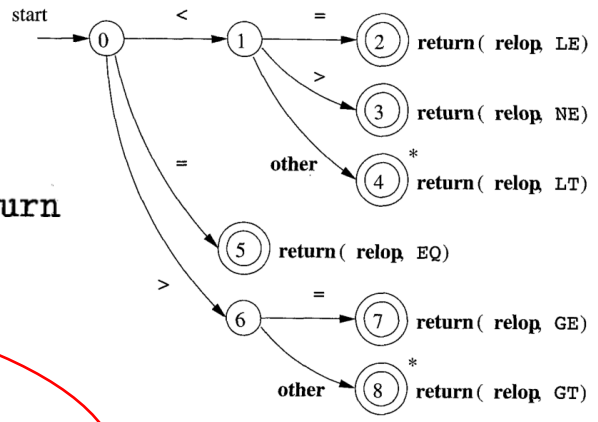


A `switch` statement based on the value of `state` takes us to the processing code

Sketch implementation of `relop` transition diagram

Building a Lexical Analyzer from Transition Diagrams

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```



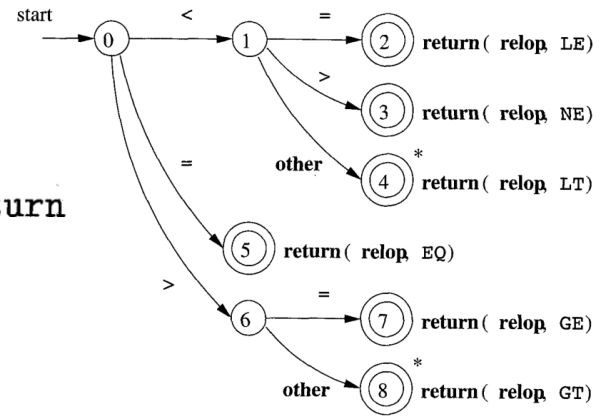
The code of a normal state:

1. Read the next character
2. Determine the next state
3. If step 2 fails, do error recovery

Sketch implementation of `relop` transition diagram

Building a Lexical Analyzer from Transition Diagrams

```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) state = 5;
                    else if ( c == '>' ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```



The code of an accepting state:

1. Perform retraction if the state has *
2. Set token attribute values
3. Return the token to parser

Sketch implementation of `relop` transition diagram

Building the Entire Lexical Analyzer

- **Strategy 1:** Try the transition diagram for each type of token sequentially
  - `fail()` resets the pointer `forward` and tries the next diagram
- **Problem:** Not efficient
  - May need to try many irrelevant diagrams whose first edge does not match the first character in the input stream

Building the Entire Lexical Analyzer

- **Strategy 2:** Run transition diagrams in parallel
  - Need to resolve the case where one diagram finds a lexeme and others are still able to process input.
  - **Solution:** take the longest prefix of the input that matches any pattern

- **Problem:** Requires special hardware for parallel simulation, may degenerate into the sequential strategy on certain machines

Building the Entire Lexical Analyzer

- **Strategy 3:** Combining all transition diagrams into one
  - Allow the transition diagram to read input until there is no possible next state
  - Take the longest lexeme that matched any pattern

- This is a commonly-adopted strategy in real-world compiler implementation (efficient & requires no special hardware)

How? Be patient ☺, we will talk about this later.

Agenda

- SPL lexical specification
- Recognizing tokens using transition diagram
- Lab task: unique valid identifiers
- Survey of language features

Lab Task: Unique Valid Identifiers

1. Define of the patterns of SPL identifiers using regular definitions/expressions
2. Design a transition diagram for identifier recognition
3. Implement the analyzer following the `getRelop()` example (no restrictions on languages)

Example

```
int global;
struct my_struct
{
  int code;
  char data;
};
int test_1_r02()
{
  struct my_struct obj;
  obj.code = global;
  global = global + 1;
  int $a = 3;
}
```

Expected output:

```
Found 5 unique IDs: global,
my_struct, code, data, obj
```

Tips:

1. Do NOT count invalid identifiers (e.g., $a)
2. Duplicates should be removed
3. Reserved words may also match the regular expressions

Please use the SPL programs at our GitHub repo to test your implementation:
https://github.com/sqlab-sustech/CS323-2024F/tree/main/lab2

Submission Requirements

- Deadline: 10:00 PM, September 22

- Please submit a zip file "stu_id.zip", which should contain your code and a readme file to tell us how to compile and run your program to analyze our provided SPL programs.

Agenda

- SPL lexical specification
- Recognizing tokens using transition diagram
- Lab task: unique valid identifiers
- Survey of language features

Aspects to Consider

- Syntax and readability
- Paradigms (e.g., procedural, object-oriented, functional, logical)
- Memory management and safety
- Community and ecosystem
- Portability and compatibility (support cross-platform development? Code can run on different operating systems and hardware architectures?)
- Performance (e.g., code execution speed & resource consumption)
- Tooling and IDE support
- Error-handling
- Security (e.g., prevention of common attacks such SQL injection)
- Application domains (e.g., for web development, for mobile apps, for machine learning , for gaming, for scientific computation, etc.)
- Learning curve (are there good learning resources?)
- Popularity and industry adoption
- …

Find Your Teammates ☺

- 【腾讯文档】CS323-2024F Project Teams https://docs.qq.com/sheet/DSlhPQUdYUkFKQ2JY?tab=BB08J2

- Please form your team by the end of the first week. Otherwise, you may miss some deadlines.