南方科技大学
SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

CS323 Lab 2

Yepang Liu

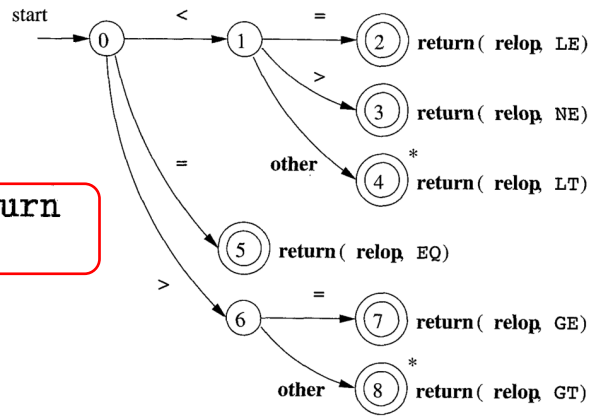
liuyp1@sustech.edu.cn

Agenda

- Lexical analysis using transition diagram
- Lab assignment 1

Building a Lexical Analyzer from Transition Diagrams

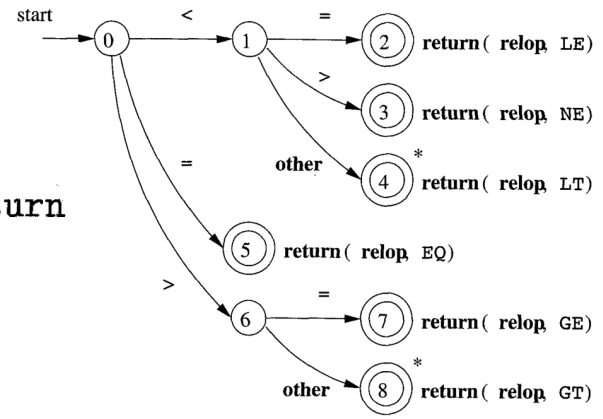
```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == '<' ) state = 1;
                    else if ( c == '=' ) 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);
        }
    }
}
```

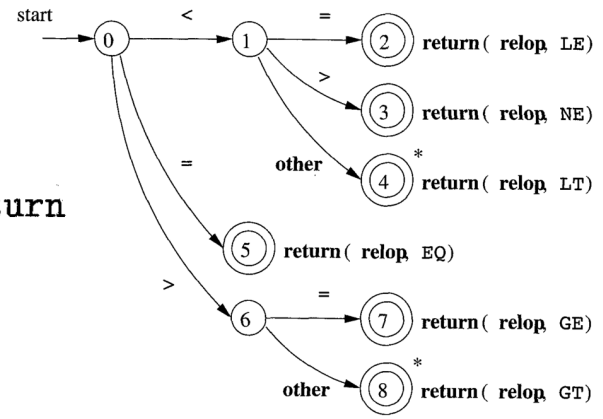


Use a variable `state` to record the current state

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);
        }
    }
}
```

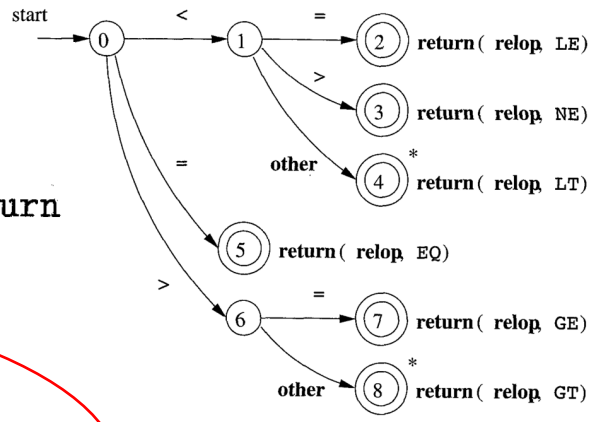


A `switch` statement based on the value of `state` takes us to the processing code

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);
        }
    }
}
```



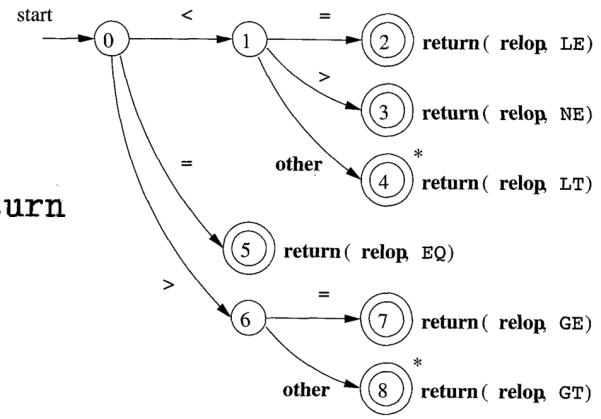
The code of a normal state:

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.

Lab Assignment 1

- Write a simplified lexical analyzer to recognize Java identifiers and reserved words
  - Define token patterns using regular definitions/expressions
  - Design transition diagrams
  - Implement the analyzer (no restrictions on languages)

Lexical Specification

- TYPE → int
- INT → /* decimal integers in a sequence of digits, which may start with the sign - */
- ID → /* identifier starting with a letter or a dollar sign or underscore and followed by any number of letters, digits, dollar signs, and underscores */
- IF → if
- ELSE → else
- WHILE → while
- RET → return
- SEMI → ;
- ASSIGN → =
- LT → <
- LE → <=
- GT → >
- GE → >=
- NE → !=
- EQ → ==
- PLUS → +
- LP → (
- RP →)
- LC → {
- RC → }

Some Tips

- How to distinguish identifiers and reserved words? Strategy 2?
- How to build the entire analyzer from separate transition diagrams? Strategy 2?
- Do not recognize == and >= as two tokens.

Test Cases

- We provide two cases on GitHub (under lab2 directory) and you are welcome to design you own test cases

```
1    int f(int x) {
2        int y=1024;
3        while (y>= 0) {
4            y = y-x;
5        }
6        if ( y > 0 ) return 1;
7        else if (y == 0) return 0;
8        else return -1;
9    }
```

Expected output:

```
TYPE ID LP TYPE ID RP LC TYPE ID ASSIGN INT
SEMI WHILE LP ID GE INT RP LC ID ASSIGN ID
MINUS ID SEMI RC IF LP ID GT INT RP RET INT
SEMI ELSE IF LP ID EQ INT RP RET INT SEMI
ELSE RET INT SEMI RC
```

```
1    int f (int x) {
2        int $y == 1024;
3        int _ = x;
4        int 3a;
5    }
```

Expected output:

```
TYPE ID LP TYPE ID RP LC TYPE ID EQ INT SEMI
TYPE ID ASSIGN ID SEMI TYPE ERROR
```

Requirements

- You can either finish the task by yourself or work together with your team members
- Deadline: 10:00 PM, September 24
- 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

Find Your Teammates ☺

- 【腾讯文档】CS323-2023秋课程项目组队 https://docs.qq.com/sheet/DSlFSU0REREVBR0pZ?tab=BB08J2