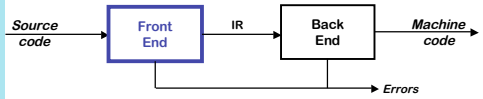
Lexical Analysis

Introduction

The Front End

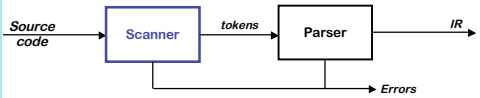


The purpose of the front end is to deal with the input language

- Perform a membership test: code ∈ source language?
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic

The Front End

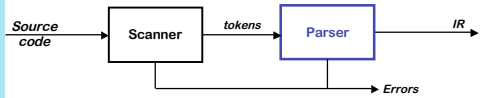


Scanner

- Maps stream of characters into words
  - Basic unit of syntax
  - x = x + y ; *becomes*
    <id,x> <eq,=> <id,x> <pl,+> <id,y> <sc,; >
- Characters that form a word are its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token type*
- Scanner discards white space & (often) comments

Speed is an issue in scanning
⇒ use a specialized recognizer

The Front End



Parser

- Checks stream of classified words (*parts of speech*) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

We'll come back to parsing in a couple of lectures

The Big Picture

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

1. *goal* → *expr*	
2. *expr* → *expr op term*	
3. | *term*	
4. *term* → number	
5. | id	
6. *op* → +	
7. | –	

S = *goal*
T = { number, id, +, - }
N = { *goal*, *expr*, *term*, *op* }
P = { 1, 2, 3, 4, 5, 6, 7}

The Big Picture

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

1. *goal* → *expr*	
2. *expr* → *expr op term*	
3. | *term*	
4. *term* → number	
5. | id	
6. *op* → +	
7. | –	

S = *goal*
T = { number, id, +, - }
N = { *goal*, *expr*, *term*, *op* }
P = { 1, 2, 3, 4, 5, 6, 7}

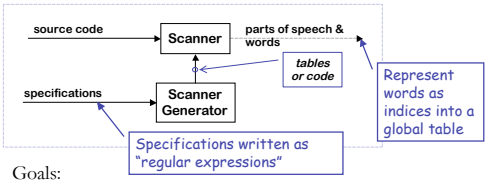
No words here!

Parts of speech, not words!

The Big Picture

Why study Lexical Analysis?

- We want to avoid writing scanners by hand
- We want to harness the theory classes



- Goals:
  - To simplify specification & implementation of scanners
  - To understand the underlying techniques and technologies

What is a Lexical Analyzer?

Source program text → Tokens

- Example of Tokens
  - Operators = + - > ({ := == <>
  - Keywords if while for int double
  - Numeric literals 43 6.035 -3.6e10 0x13F3A
  - Character literals 'a' '~' '\"'
  - String literals "6.891" "Fall 06" "\"\"" = empty"
- Example of non-tokens
  - White space space(' ') tab('\t') end-of-line('\n')
  - Comments /*this is not a token*/

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

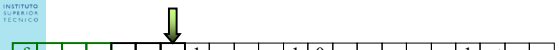
for

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for

Lexical Analyzer in Action

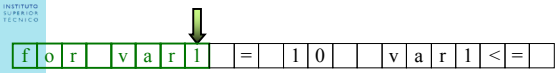


for

Lexical Analyzer in Action

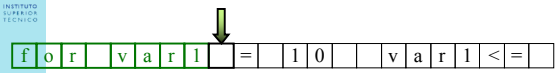
for

Lexical Analyzer in Action



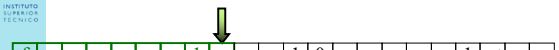
for ID("var1")

Lexical Analyzer in Action



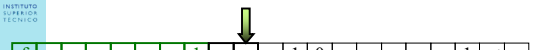
for ID("var1")

Lexical Analyzer in Action



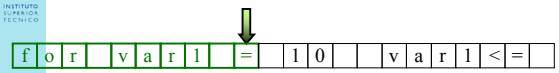
for ID("var1")

Lexical Analyzer in Action



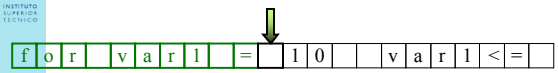
for ID("var1")

Lexical Analyzer in Action



for ID("var1") eq_op

Lexical Analyzer in Action



for ID("var1") eq_op

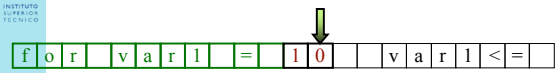
Lexical Analyzer in Action

for ID("var1") eq_op

Lexical Analyzer in Action

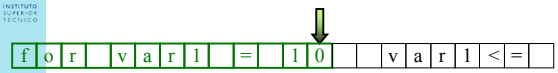
for ID("var1") eq_op

Lexical Analyzer in Action



for ID("var1") eq_op

Lexical Analyzer in Action



for ID("var1") eq_op Num(10)

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID(“var1”) eq_op Num(10)

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID(“var1”) eq_op Num(10)

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID(“var1”) eq_op Num(10)

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID(“var1”) eq_op Num(10)

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID("var1") eq_op Num(10)

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID("var1") eq_op Num(10)

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID("var1") eq_op Num(10)

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID("var1") eq_op Num(10)

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID("var1") eq_op Num(10) ID("var1")

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

for ID("var1") eq_op Num(10) ID("var1")

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

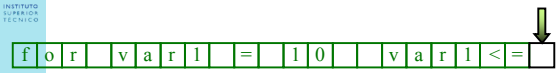
for ID("var1") eq_op Num(10) ID("var1")

Lexical Analyzer in Action

f	o	r		v	a	r	1		=		1	0			v	a	r	1	<	=	

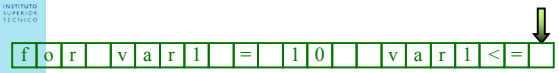
for ID("var1") eq_op Num(10) ID("var1") leq_op

Lexical Analyzer in Action



for ID("var1") eq_op Num(10) ID("var1") leq_op

Lexical Analyzer in Action



for ID("var1") eq_op Num(10) ID("var1") leq_op

Lexical Analyzer needs to…

- Partition Input Program Text into Subsequence of Characters Corresponding to Tokens
- Attach the Corresponding Attributes to the Tokens
- Eliminate White Space and Comments

Lexical Analysis: Basic Issues

- How to Precisely Match Strings to Tokens
- How to Implement a Lexical Analyzer

Regular Expressions

Lexical patterns form a *regular language*

**** any finite language is regular ****

Regular expressions (REs) describe regular languages

Ever type "rm *.o a.out" ?

Regular Expression (over alphabet Σ)

- ε is a RE denoting the set {ε}
- If a is in Σ, then a is a RE denoting {a}
- If *x* and *y* are REs denoting *L(x)* and *L(y)* then
  - *x* | *y* is an RE denoting *L(x)* ∪ *L(y)*
  - *xy* is an RE denoting *L(x)L(y)*
  - x^* is an RE denoting *L(x)**

Precedence is closure, then concatenation, then alternation

Set Operations (review)

Operation	Definition
Union of L and M ***Written L ∪ M***	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation of L and M ***Written LM***	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure of L ***Written L****	$L^* = \cup_{0 \le i \le \infty} L^i$
Positive Closure of L ***Written L+***	$L^+ = \cup_{1 \le i \le \infty} L^i$

These definitions should be well known

Examples of Regular Expressions

Identifiers:

Letter → (a|b|c| … |z|A|B|C| … |Z)
Digit → (0|1|2| … |9)
Identifier → *Letter* (*Letter* | *Digit*)*

Numbers:

Integer → (+|-|ε) (0| (1|2|3| … |9)(*Digit**))
Decimal → *Integer* . *Digit**
Real → (*Integer* | *Decimal*) E (+|-|ε) *Digit**
Complex → (*Real* , *Real*)

Numbers can get much more complicated!

Regular Expressions (the point)

Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions

Some of you may have seen this construction for string pattern matching

⇒ We study REs and associated theory to automate scanner construction !

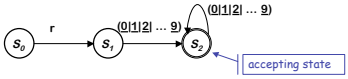
Example

Consider the problem of recognizing Register names

Register → r (0|1|2| … | 9) (0|1|2| … | 9)*

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)



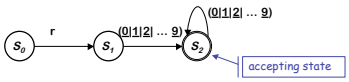
Recognizer for *Register*

Transitions on other inputs go to an error state, s_e

Example (continued)

DFA operation

- Start in state S_0 & take transitions on each input character
- DFA accepts a word x *iff* x leaves it in a final state (S_2)



Recognizer for *Register*

So,

- r17 takes it through s_0, s_1, s_2 and accepts
- r takes it through s_0, s_1 and fails
- a takes it straight to s_e

Example (continued)

To be useful, recognizer must turn into code

```
Char ← next character
State ← s0
while (Char ≠ EOF)
    State ← δ(State,Char)
    Char ← next character
if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4,5 ,6,7,8,9	*All others*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table encoding RE

Example (continued)

To be useful, recognizer must turn into code

```
Char ← next character
State ← s0
while (Char ≠ EOF)
    State ← δ(State,Char)
    perform specified action
    Char ← next character
if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4,5 ,6,7,8,9	*All others*
s_0	s_1 *start*	s_e *error*	s_e *error*
s_1	s_e *error*	s_2 add	s_e *error*
s_2	s_e *error*	s_2 add	s_e *error*
s_e	s_e *error*	s_e *error*	s_e *error*

Table encoding RE

What about a Tighter Specification?

r *Digit Digit** allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31 ?

Write a tighter regular expression

- *Register* → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))
- *Register* → r0|r1|r2| … |r31|r00|r01|r02| … |r09

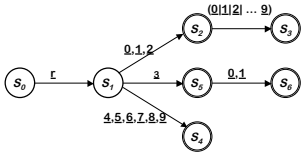
Produces a more complex DFA

- Has more states
- Same cost per transition
- Same basic implementation

Tighter Register Specification (cont'd)

The DFA for

Register → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))



- Accepts a more constrained set of registers
- Same set of actions, more states

Tighter Register Specification (cont'd)

δ	r	0,1	2	3	4-9	All other s
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e

Runs in the same skeleton recognizer

Table encoding RE for the tighter register specification

Summary

- The role of the lexical Analyzer
  - Partition input stream into tokens
- Regular Expressions
  - Used to describe structure of tokens
- DFA: Deterministic Finite Automata
  - Machinery to recognize Regular Languages