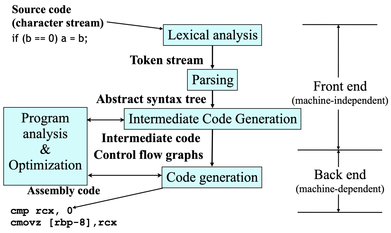
Lexical Analysis

Bigger picture



This lecture:

ad-hoc lexing
regular expressions (review)
using lexer generator
how lexer generators work behind the scene (review)

Goal: Break input stream into tokens

Tokens

integer literal 2
identifier count
keyword while
separator (
string literal "hello"
operator ++

kept: identifiers, keywords, literals, operator
discarded: comments, whitespace

every stage of a compiler is going to forget some information that's not important for the rest of the compiler

token := token type + attributes

Identifier ("count")

token := token type + attributes + location info (file, line, col)

How to lex/scan/tokenize?

Example: scan an identifier

a novice programmer writes this code:

```
while (true) {
   int c = input.read()
   if (!isIdChar(c)) break;
   id.append( (char) c );
}
```

What are the problems?

Problem 1: if c is not an id char, then c is consumed from the input and is not restored.

Problem 2: overlapping tokens

White
while

To solve problem 1:

```
while (isIdChar(input.peek()) {
   int c = input.read()
   id. append( (char) c )
}
```

(peek unavailable in java standard library!)

To solve problem 2: backtracking

Regular expressions

a more declarative approach to constructing lexers

Idea:
- Define legal tokens using REs (spec)
- Synthesize lexer from this spec
- Declarative

Common theme in this course: use mathematical formalism to make the compiler writer's life easier

Regular expressions denote a set of strings

L(R) = "language of R"

R,S	L(R)
a	{ "a" }
ε	{ "" }
R|S (or R+S)	L(R) ∪ L(S)
RS (or R · S)	{ rs | r ∈ L(R) ∧ s ∈ L(S) }
R*	L(ε) ∪ L(R) ∪ L(RR) ∪ ...

Kleene star

a | bc* = a | (b (c*))

R? := R | ε

R+ := R · R*

[abc] = [a-c] := a | b | c

[^ab] = any character but a, b

. = [^\n]

[^] = any character

Lexer generators:

flex jflex ocamllex ...

Input: RE + action, for each token type
Output: code that invokes action on matched token

Actions:
- create & return token
- discard (irrelevant to subsequent phases of the compiler)
- raise error
- side effects??

Abbreviations:

```
digit := [0-9]              // define digit
posint := [1-9][digit]* // use digit
```

no recursion!

```
id := [a-z] | [a-z] {id} // not allowed
```

recursion => CFG, overkill for lexical analysis

Tokenizing multiple RE's

elsex = 0

else | x = 0
else x | = 0

two possible tokenizations:

What does Java do?

Rule used in most PLs: longest matching token (scan to the right a.m.a.p. while maintaining a valid token)

"Longest matching token" rule can lead to tokenization that will fail in a later compiler stage but would otherwise not fail if "longest matching" rule is not in effect:

a++b
a+ +b

What should your Joos compiler do? ++ is not supported in Joos

scan Java keywords/operators not supported in Joos 1W—but only to reject them

List<List<Integer>>

>>

>␣>

Lexer states

Can condition rules on lexer state.

lexer action: { yybegin(state); } // enter state