CSC 372, Spring 2025

Lexing and Recursive Descent Parsing

Michelle Strout

February 4, 2025

Plan

- **Announcements**
  - LA1 is posted and due Friday Feb 14th
  - Tomorrow will be posting SA3, which will be an online MT1 review
- **Last time**
  - Grammar clarification
  - Tokenization/Lexing as a part of syntactic analysis
  - Algebraic datatypes in SML
- **Today**
  - Show Anki deck of PL concepts
  - TopHat Questions and ICA4: Quiz on SML concepts from SA2
  - Tokenization/Lexing as a part of syntactic analysis
  - Algebraic datatypes in SML

TopHat Questions

- Functional languages and mutability question
- Link languages to motivation in TopHat
  - Fortran
  - Erlang
  - Haskell
  - SML
  - Ada

ICA4: Quiz on SML concepts

- Read the instructions on the quiz

Reprise: How should I study for 372?

- **Gather and create example questions**
  - Questions from class slides, TopHat, quizzes.
  - Recall practice questions at end of Ray Toal readings.
  - Exercises in "ML for the Working Programmer" readings.
  - Create questions and answers about all the concepts that are in the class slides.
  - Create questions and answers about the code you are writing.
- **Dig in and work on understanding and learning**
  - Collaborate with others and AIs to create questions and formulate answers.
  - Verify things you are unsure about by looking for alternative sources of information on the web such as books, other course material, or asking on piazza.
- **Use spaced repetition to study (show Anki)**

Outline for rest of today

- Regular expressions to specify tokens
- Complications with tokenization/lexing
- Simple tokenization for PA1
- Recursive descent parsing for PA1
- SML hints: debugging with print, patterns in val bindings

Learning by doing in LA1 (show face.svg)

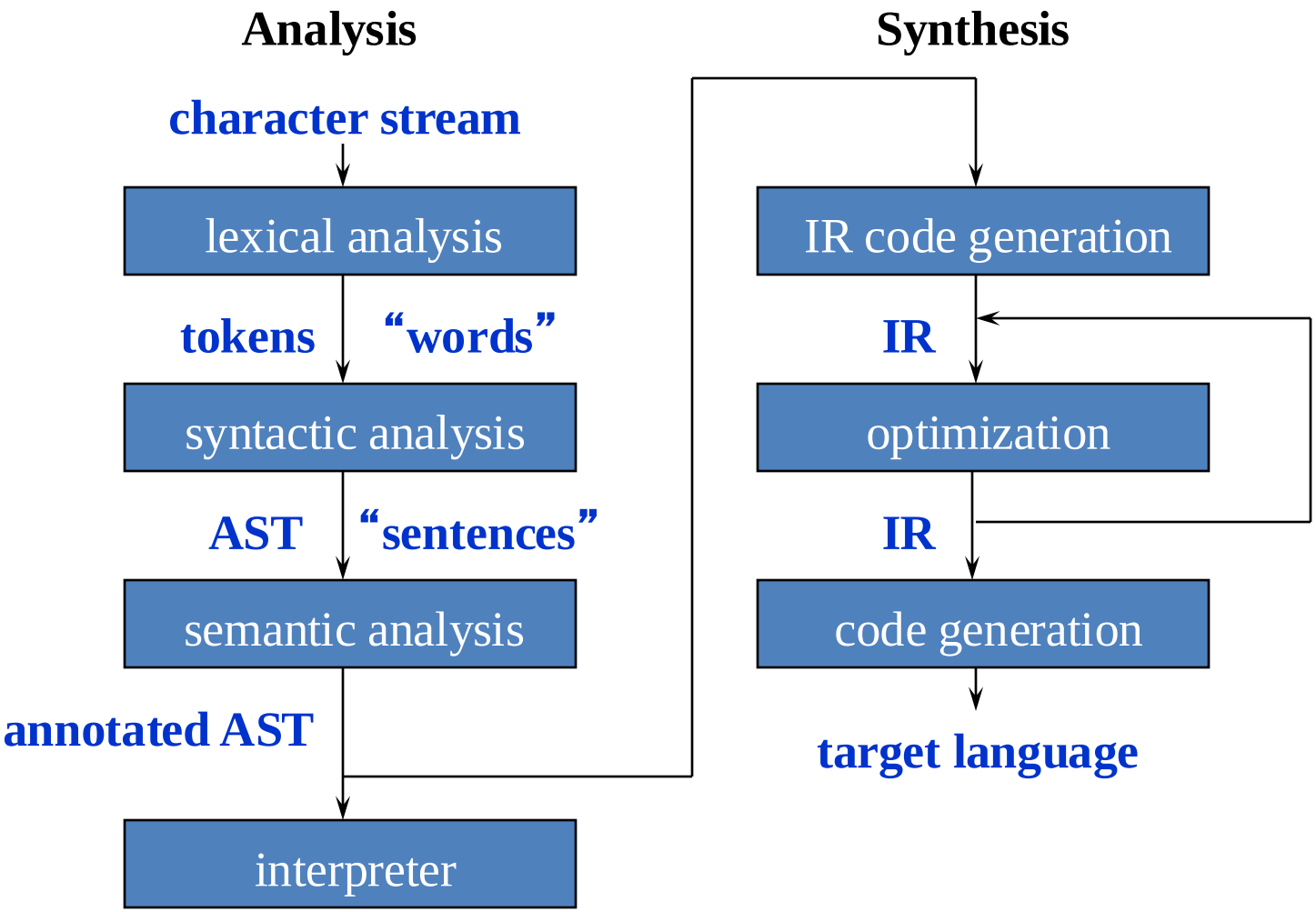
- Large assignment 1 is a compiler from shapes to svg

```
CIRCLE 120 150 60 white
```

```
<circle cx="120" cy="150" r="60" fill="white" />
```

- Going to specify the input with a context free grammar and tokens with regular expressions
- Going to implement a lexer/tokenization, parser, AST (abstract syntax tree), and "codegen" in SVG

Structure of a Typical Compiler



Regular operators

choice:	**A | B**	**a string from L(A) or from L(B)**
concatenation:	**A B**	**a string from L(A) followed by a string from L(B)**
repetition:	**A***	**0 or more concatenations of strings from L(A)**
	A^+	**1 or more**
grouping:	**(A)**	

Concatenation has precedence over choice: A|B C vs. (A|B)C

More syntactic sugar, used in scanner generators:

[abc] means a or b or c

[\t\n] means tab, newline, or space

[a-z] means a,b,c, …, or z

Example Regular Expressions and Regular Definitions

Regular definition:

name : regular expression

name can then be used in other regular expressions

Keywords "print", "while"

Operations: "+", "-", "*", "(", ")"

Identifiers:

let : [a-zA-Z] // chose from a to z or A to Z

dig : [0-9]

id : let (let | dig)*

Numbers: dig^+ = dig dig*

What are the regular expressions for “shapes” tokens?

```
CIRCLE 120 150 60 white
LINE 0 0 300 300 black
RECTANGLE 30 20 300 250 blue
```

```
datatype token =
    TokenCIRCLE
  | TokenLINE
  | TokenRECTANGLE
  | TokenNUM of int
  | TokenCOLOR of string
```

```
TokenCIRCLE:
TokenLINE:
TokenRECTANGLE:
TokenNUM:
TokenCOLOR:
```

Outline for today

- Regular expressions to specify tokens
- Complications with tokenization/lexing
- Simple tokenization for PA1
- Recursive descent parsing for PA1
- SML hints: debugging with print, patterns in val bindings

Complications

1. "1234" is an NUMBER but what about the "123" in "1234" or the "23", etc. Also, the scanner/lexer must recognize many tokens, not one. It should only stop at end of file.

2. "if" is a keyword or reserved word IF, but "if" is also defined by the reg. exp. for identifier ID. We want to recognize IF.

3. We want to discard white space and comments, most of the time.

4. "123" is a NUMBER but so is "235" and so is "0", just as "a" is an ID and so is "bcd", we want to recognize a token, but add attributes to it.

Complications 1

1. "1234" is an NUMBER but what about the "123" in "1234" or the "23", etc. Also, the scanner/lexer must recognize many tokens, not one. It should only stop at end of file.

So:

recognize the largest string defined by some regular expression,

only stop getting more input if there is no more match. This introduces the need to reconsider a character, as it is the first of the next token

e.g. *fname(a,bcd);*

would be scanned as

ID OPENPAREN ID COMMA ID CLOSE SEMI EOF

scanning *fname* would consume "(", which would be put back and then recognized as OPENPAREN

Related: What if "0" is an acceptable token, but "07" isn't?

Complication 2

2. "if" is a keyword or reserved word IF, but "if" is also defined by the reg. exp. for identifier ID, we want to recognize IF, so

Have some way of determining which token (IF or ID) is recognized.

This can be done using priority, e.g. in scanner/lexer generators an earlier definition has a higher priority than a later one.

By putting the definition for IF before the definition for ID in the input for the scanner generator, we get the desired result.

What about the string "ottersarecute"?

Complication 3

3. we want to discard white space and comments and not bother the parser with these. So:

in scanner/lexer generators, we can

specify, using a regular expression, white space e.g. [\t\n]

and return no token, i.e. move to the next character

specify comments using a (ICKY) regular expression and again return no token, move to the next

Complication 4

4. "123" is a NUMBER but so is "235" and so is "0", just as "a" is an ID and so is "bcd", we want to recognize a token, but add attributes to it. So,

Scanners/Lexers return Symbols, not tokens. Different in SML?
A Symbol is a (token, tokenValue) pair,
e.g. (NUMBER,123) or (ID,"a").

Often more information is added to a symbol, e.g. line number and position.

Revisit: the regular expressions for “shapes” tokens?

```
CIRCLE 120 150 60 white
LINE 0 0 300 300 black
RECTANGLE 30 20 300 250 blue
```

```
datatype token =
    TokenCIRCLE
  | TokenLINE
  | TokenRECTANGLE
  | TokenNUM of int
  | TokenCOLOR of string
```

```
TokenCIRCLE:
TokenLINE:
TokenRECTANGLE:
TokenNUM:
TokenCOLOR:
```

Simple Tokenization for PA1

- LA1: compiler from shapes to SVG

```
CIRCLE 120 150 60 white
```

```
<circle cx="120" cy="150" r="60" fill="white" />
```

- Just because something can be complicated doesn't mean it always is
- Ask an AI or search engine how to break up a string into substrings using whitespace as a delimeter
- What if we were going from SVG to shapes?

Outline for today

- Regular expressions to specify tokens
- Complications with tokenization/lexing
- Simple tokenization for PA1
- **Recursive descent parsing for PA1**
- **SML hints: debugging with print, patterns in val bindings**

Predictive Parsing

- Predictive parsing, such as recursive descent parsing, creates the parse tree TOP DOWN, starting at the start symbol, and doing a LEFT-MOST derivation.
- For each non-terminal N there is a function recognizing the strings that can be produced by N, with one (case) clause for each production.

- Consider:

```
start     -> stmts EOF
stmts     ->  ε | stmt stmts
stmt      -> ifStmt |  whileStmt | ID = NUM
ifStmt    -> IF id { stmts }
whileStmt -> WHILE id { stmts }
```

- Draw parse tree on white board for this

```
WHILE x { IF y { z = 42 } }
```

Predictive Parser: Recursive Descent


```
start     -> stmts EOF
stmts     -> ε | stmt stmts
stmt      -> ifStmt |  whileStmt
ifStmt    -> IF id { stmts }
whileStmt -> WHILE id { stmts }
```

```
void start() { switch(m_lookahead) {
   case IF, WHILE, EOF: stmts(); match(Token.Tag.EOF); break;
   default:   throw new ParseException(…);
}}
void stmts() { switch(m_lookahead) {
   case IF,WHILE:  stmt(); stmts(); break;
   case EOF:      break;
   default:       throw new ParseException(…);
}}
void stmt() { switch(m_lookahead) {
   case IF:   ifStmt();break;
   case WHILE: whileStmt(); break;
   default:   throw new ParseException(…);
}}
void ifStmt() {switch(m_lookahead) {
   case IF: match(id); match(OPENBRACE);
         stmts(); match(CLOSEBRACE); break;
   default: throw new ParseException(…);
}}
```

Recursive Descent Parsing

- Each non-terminal becomes a function
  - that mimics the RHSs of the productions associated with it
  - and choses a particular RHS:
    - an alternative based on a look-ahead symbol
  - and throws an exception if no alternative applies
- When does this NOT work?

Determine Look Ahead per grammar rule

Grammar Rule	Lookahead Token
start -> stmts EOF	// no lookahead
stmts -> ε	
stmts -> stmt stmts	
stmt -> ifStmt	
stmt -> whileStmt	
stmt -> ID = NUM	
ifStmt -> IF id { stmts }	
whileStmt -> WHILE id { stmts }	WHILE

Look at LA1 starter code

- Observations

Outline for today

- Regular expressions to specify tokens
- Complications with tokenization/lexing
- Simple tokenization for PA1
- Recursive descent parsing for PA1
- **SML hints: debugging with print, patterns in val bindings**

SML hints: debugging with print

- Sequence expressions
  - Value of whole expression is value of last item in a semicolon separated sequence
  - All expressions but the last need to be of type unit
- Combined with print, this can be used for debugging

```
let val x=3
in (print "x="; print (Int.toString x); print "\n"; x)
end;
```

- What is the type of 'print' in SML?

SML hints: patterns in val bindings

- Patterns can be used in ...
  - Function clauses
  - Case expressions
  - And Val bindings

```
val (x,y) = partition (fn x => true) [5,4,3]
```