CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Fall 2023

Scanner Generators

- Efficient lexical-analyzer generators exist which can translate regular expressions → DFAs, and produce tokens for a given input stream.

- Lex Scanner generator:
  - Classic scanner generator, for the C Language, developed at AT&T Bell Labs
  - produces an entire scanner module coded in C
  - saves a great deal of effort when programming a scanner
  - many low-level details (reading characters effectively, matching against token definitions) are already programmed
  - programmer only needs to write the *Scanner Specification*

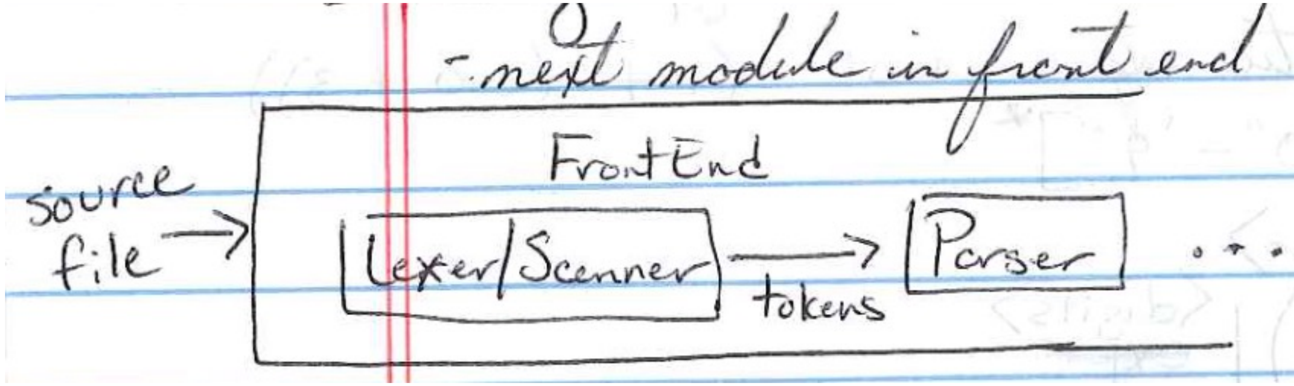
Other Scanner Generators

- Other Scanner generators also exist: Lex, Flex (reimplementation of Lex that produces faster and more reliable Scanners), JFlex (Java Scanner instead of C), JavaCC
- We'll use **JavaCC** this semester

Front End of a Compiler – Road Map

- Scanner (Lexical Errors)
- Parser (Syntax Errors)
- Semantic Analyzer (Semantic Errors)

Parsing



Phase	Input	Output
Lexer	String of Characters	Seq. of Tokens
Parser	Seq. of Tokens	Parse Tree

Regular languages are "weak"

- Many languages can't be expressed using **regular languages** or **finite automata**.
- *Example:* "the set of all balanced parentheses"

$$(^i)^i \quad i \geq 0$$

( )

(( ))

((( )))

The “balanced” characteristic is found in many P/L constructs

```
(1 * (250 + 3))
```

Bash scripting:

```
if

…

fi
```

- Cannot be expressed by regular languages.
- Can't count arbitrary high w/ a finite set of states.
- But can represent "parity":
  - e.g. *an even number of 1’s*
  - Regular languages can represent a "mod k" problem.
- Going to use a different formalism for parsing.

CFGs: Context Free Grammars

- Need a language for describing valid sequences of tokens that compose a legal program (in Ram, Java, C++, etc.).
- Most useful programming languages have a *recursive structure*.
- Will need something more powerful than finite automata to parse languages described by grammars.

Example

- Representing some *programming language constructs*:

```
Exp → Exp [Exp]
Exp → Exp op Exp
```

- *Note:* Parsing grammar won't detect all errors. There is still a type checking phase that will run afterwards.

CFG Components

Defined by the 4-tuple:

G=(T,N,P,S)

- set of <u>terminals</u>: T
- set of <u>nonterminals</u>: N
- <u>start symbol</u>: S
- set of <u>productions</u>:
  - $X \to Y_1 \ldots Y_N$
  - $X \in N$
  - $Y_i \in N, T, \epsilon$

Example: strings of balanced parentheses

- Productions:

  `S → ( S )`

  `S → ϵ`

- <u>Set of Terminals:</u> `{ (, ) }`
- <u>Set of Nonterminals:</u> `{S}`
- <u>Start symbol:</u> first production if not explicitly mentioned.

Productions can be read as replacement rules.

Derivations

$$S \rightarrow (\; S \;)$$
$$S \rightarrow \epsilon$$

Example Derivation: $S \rightarrow (S) \rightarrow ((S)) \rightarrow (((S))) \rightarrow$ "((()))"

In a derivation (sequence of productions):

1. begin with a string with only the start symbol, S
2. replace any non-terminal X in the string by the RHS of some production: $X \rightarrow Y_1 \ldots Y_n$
3. repeat (2) until there are no non-terminals remaining

- In programming language grammars, the terminals are the tokens of the language.
- Once generated, terminals are permanent in the string: there are no rules for going backwards, and no other rules for replacing them.
- Derivations can also be drawn as a tree:
  - start symbol is the tree's root
  - for production $X \rightarrow Y_1 \ldots Y_n$, add children $Y_1 \ldots Y_n$ to node X

Example

Grammar:

```
E -> E + E
   | E * E
   | ( E )
   | id
```

Input String:

```
id * id + id
```

- Is the string in the language of this grammar?
- Build a derivation and the corresponding parse tree.
- What are the interesting properties of parse trees?

Different derivations of the same string are often possible

- <u>Leftmost derivation:</u> the leftmost non-terminal is always the next expanded symbol.
- <u>Rightmost derivation:</u> right terminal is the next to be expanded.
- Also possible: <u>neither leftmost or rightmost</u>.

Example: rightmost derivation

- Leftmost and rightmost derivations may produce the same parse tree.

- Single parse tree could have many different derivations.
- We are not only interested in whether $s \in L(G)$, but we also want to know the parse tree for s!

Example

Which of these strings are in the language of the CFG?

CFG:

```
S → a X a
X → b Y
  | ε
Y → c X c
  | ε
```

A. "aba"
B. "abba"
C. "abcba"
D. "abcbcba"

After the Break

How did I know which production rules to use when I expand a non-terminal?

Before the Break

- Left off with the question of "how did I know which production rule to use when I expanded a non-terminal?”

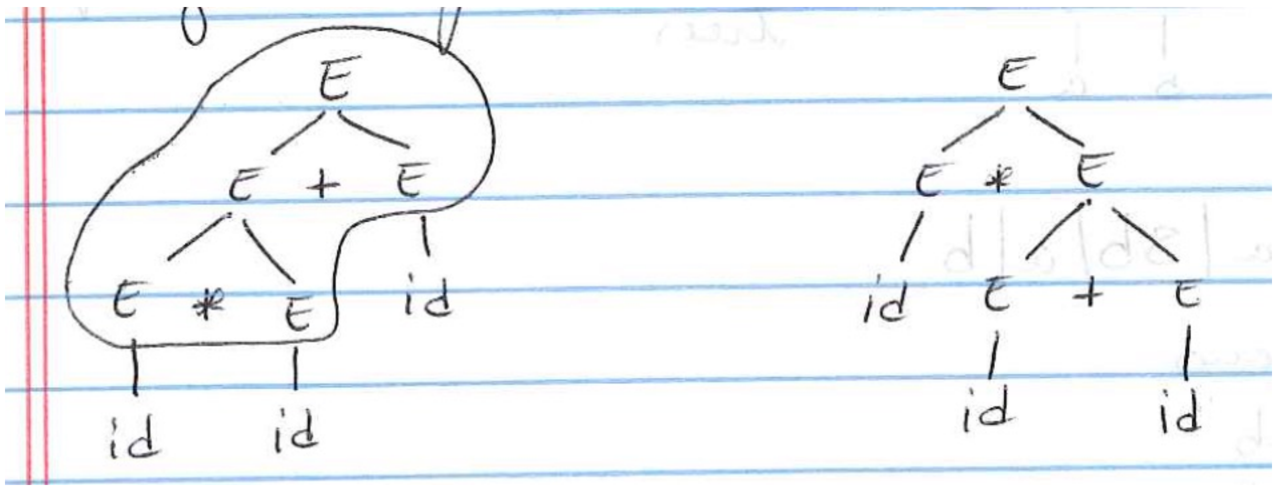
Grammar: `E → E+E | E*E | (E) | id`

Input string: `id * id + id`

- We produced an identical parse tree using a <u>leftmost derivation</u> and a <u>rightmost derivation</u>.

Ambiguity

- A grammar is <u>ambiguous</u> if it has more than one parse tree for the same string.



Is ambiguity bad?

Example

- Is this an ambiguous grammar?
- `S → SS | a | b`
- *Thought process:* try to construct an input string that could yield different parses
- Try "`bab`"

Example

- Is this an ambiguous grammar?
- $S \rightarrow Sa \mid Sb \mid a \mid b$
- *Input string:* “bab”

Handling ambiguity

- The most direct method is to rewrite the grammar so that it is <u>unambiguous</u>.

Ambiguous Grammar:

```
E → id | num | E * E | E / E | E + E | E - E | (E)
```

Unambiguous Grammar:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → id | num | (E)
```

Big Idea: introducing *new* non-terminals to enforce precedence

- Let E handle the addition and subtraction.
  - (the operators that we want to perform last, lowest precedence, shallowest in parse tree)
- Let T handle multiplication and division.

Parse Tree Example

Unambiguous Grammar:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → id | num | (E)
```

Input String:

```
id * id + id
```

Example

- Select the unambiguous grammar that is <u>equivalent</u> to this ambiguous grammar.

Ambiguous Grammar:

$S \to SS \mid a \mid b$

Unambiguous Grammars:

a) $S \to Sa \mid Sb \mid \epsilon$

b) $S \to SS' \mid a \mid b$
$S' \to a \mid b$

c) $S \to S \mid S'$
$S' \to a \mid b$

d) $S \to Sa \mid Sb$

Predictive Parsing: Recursive Descent Parsing

Our first parsing algorithm!

Recursive descent:

- top-down parsing algorithm
  - (there are also bottom-up algorithms that we'll explore later)
- Parse tree is constructed *from the top*, and *left-to-right*.

Idea: given a grammar and a token stream...

1. Build a parse tree starting with the top-level nonterminal, and trying the production rules *in order*.
2. Backtrack if necessary.

Example

Grammar:

```
E → T | T + E
T → int | int * T | (E)
```

Token Stream:

```
<LPAREN , "("> ,   <INT , "5"> ,   <RPAREN , ")">
```

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$S \rightarrow \text{begin } S\ L$$
$$S \rightarrow \text{print } E$$

$$L \rightarrow \text{end}$$
$$L \rightarrow ;\ S\ L$$

$$E \rightarrow \text{num} = \text{num}$$

GRAMMAR 3.11.

```
final int IF=1, THEN=2, ELSE=3, BEGIN=4, END=5, PRINT=6,
          SEMI=7, NUM=8, EQ=9;

int tok = getToken();

void advance() {tok=getToken();}
void eat(int t) {if (tok==t) advance(); else error();}

void S() {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S();
                    eat(ELSE); S(); break;
        case BEGIN: eat(BEGIN); S(); L(); break;
        case PRINT: eat(PRINT); E(); break;
        default:    error();
       }}
void L() {switch(tok) {
        case END:   eat(END); break;
        case SEMI:  eat(SEMI); S(); L(); break;
        default:    error();
       }}
void E() {  eat(NUM); eat(EQ); eat(NUM); }
```

Recursive Descent Parsing Alg:

- *Issue:* Only works on grammars where the first *terminal* symbol of each production provides enough information about which production to choose.
- Does not work on the grammar:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow \mathrm{id} \mid \mathrm{num} \mid (E)$$

Could rewrite grammar...

```
E → E + T | E - T | T
T → T * F | T / F | F
F → id | num | (E)
```

... E→E+T *becomes* E→T+E

- Topic of Left Recursion will be discussed next class.
- Also need to implement back-tracking.

Backtracking Implementation

Grammar:

```
E → T | T + E
T → int | int * T | (E)
```

Big idea: define boolean functions that return success.

To run:

- initialize `next` to point to the first token
- invoke `E()`
  - if string is in the grammar, will return true, else will return false

```
boolean advance(int t) { return next++ == t; }

boolean E_1() { return T(); }
boolean E_2() { return T() && advance(PLUS) && E(); }
                     // will have side-effects that bumps our token pointer past T
                     // requires short circuiting

boolean E() { savepos = next; return E_1() || (next = savepos, E_2()); }

boolean T_1() { return advance(INT); }
boolean T_2() { return advance(INT) && advance(TIMES) && T(); }
boolean T_3() { return advance(LPAREN) && E() && advance(RPAREN); }
boolean T() { savepos = next; return T_1()
                                || (next = savepos, T_2())
                                || (next = savepos, T_3()) ; }
                   // local savepos variable in case of backtracking
                   // restore pointer in expression sequence in case T_1()
                   // had any side-effects when we tried it
```

Next Week

- Left Recursion