Linguaggi di Programmazione (Analisi Lessicale e Sintattica)

Lucidi basati su materiale
in inglese di Ras Bodik
http://inst.eecs.berkeley.edu/~cs164/fa04

Three kinds of execution environments

- Interpreters
  - Scheme, lisp, perl
  - popular interpreted languages later got compilers
- Compilers
  - C
  - Java (compiled to bytecode)
- Virtual machines
  - Java bytecode runs on an interpreter
  - interpreter often aided by a JIT compiler

The Structure of a Compiler

1. Scanning (Lexical Analysis)
2. Parsing (Syntactic Analysis)
3. Type checking (Semantic Analysis)
4. Optimization
5. Code Generation

The first 3, at least, can be understood by analogy to how humans comprehend English.

Lexical Analysis

- Lexical analyzer divides program text into “words” or “tokens”

```
if x == y then z = 1; else z = 2;
```

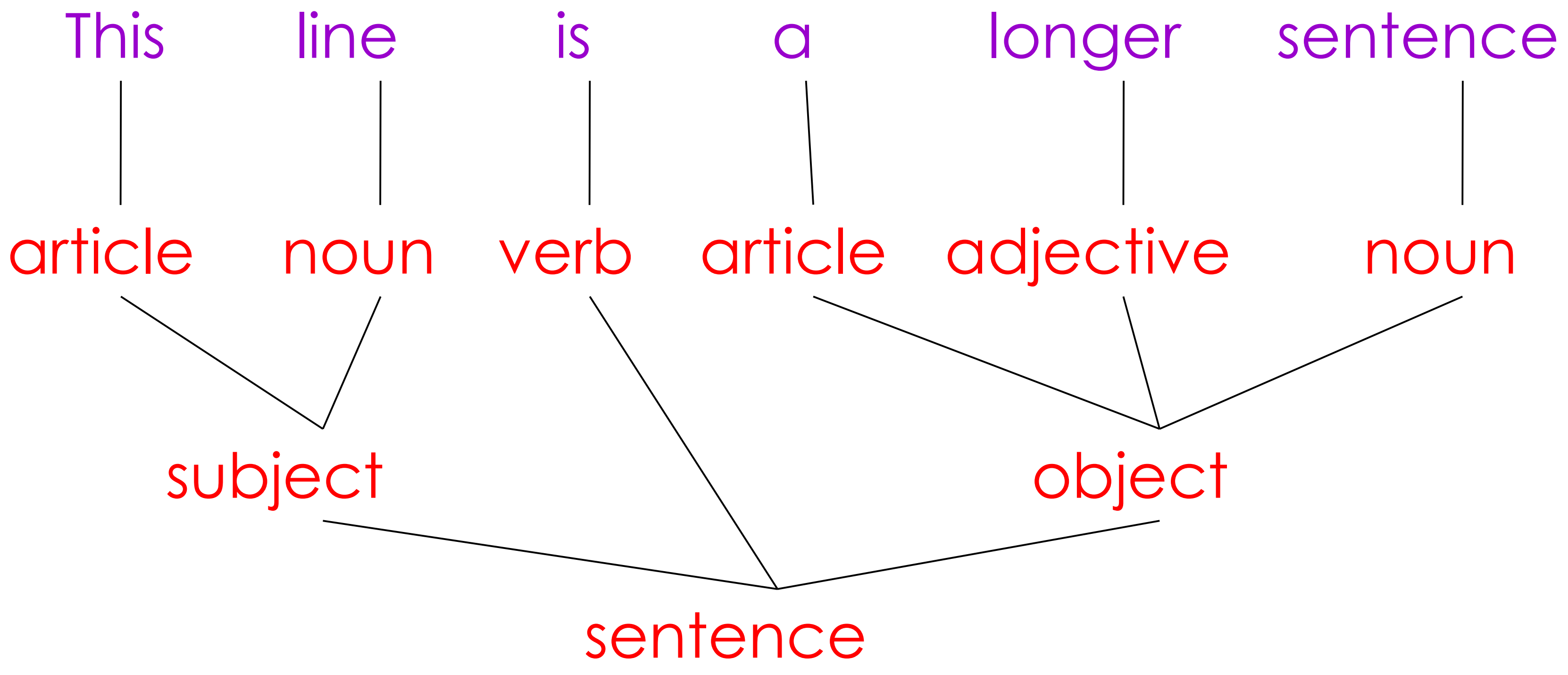
- Units:

```
if, x, ==, y, then, z, =, 1, ;, else, z, =, 2, ;
```

Parsing

- Once words are understood, the next step is to understand sentence structure
- Parsing = Diagramming Sentences
  - The diagram is a tree

Diagramming a Sentence

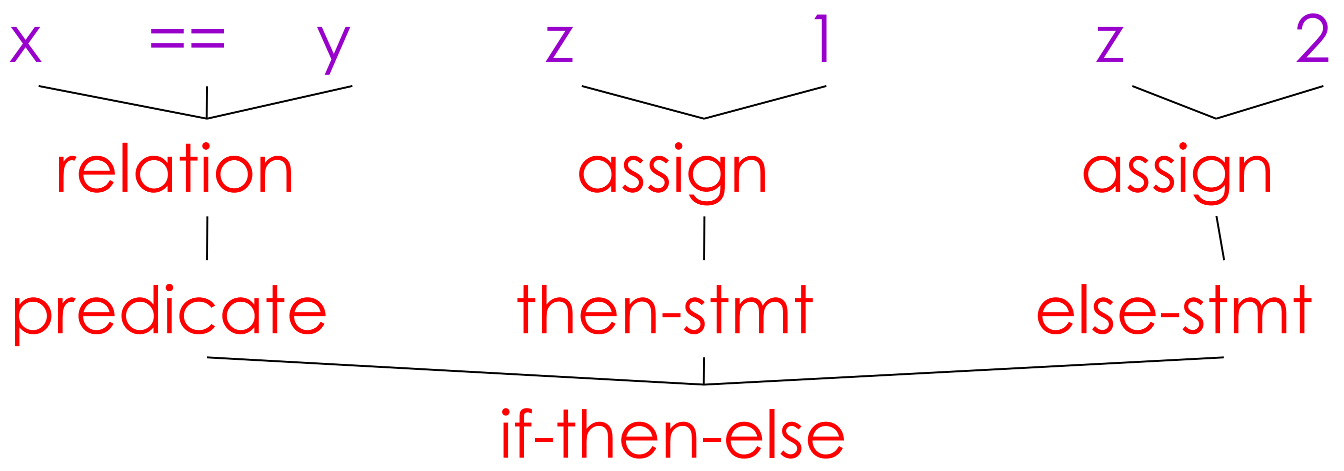


Parsing Programs

- Parsing program expressions is the same
- Consider:

  If x == y then z = 1; else z = 2;

- Diagrammed:



Semantic Analysis in English

- Example:

  Jack said Jerry left his assignment at home.

  What does “his” refer to? Jack or Jerry?

- Even worse:

  Jack said Jack left his assignment at home?

  How many Jacks are there?

  Which one left the assignment?

Semantic Analysis I

- Programming languages define strict rules to avoid such ambiguities

- This *C/C++* code prints “4”; the inner definition is used

```
{
   int Jack = 3;
   {
      int Jack = 4;
      printf("%d", Jack);
   }
}
```

Semantic Analysis II

- Compilers also perform checks to find bugs

- Example:

  Jack left her homework at home.

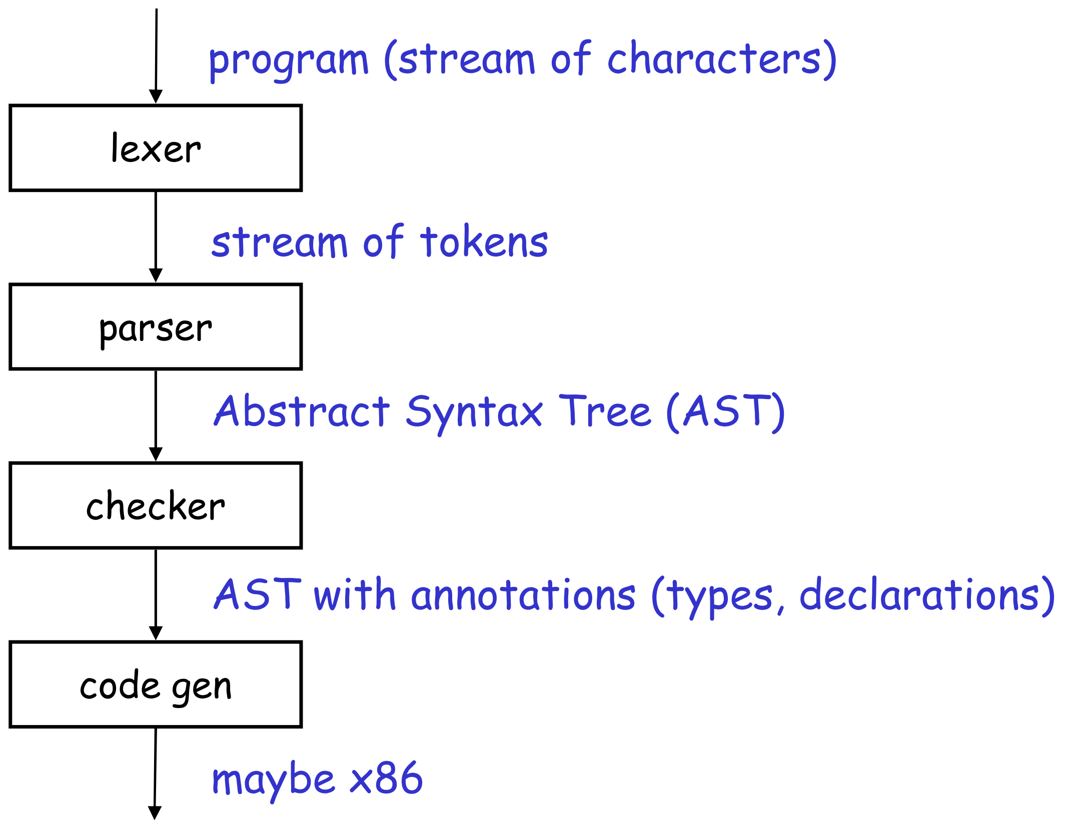
- A “type mismatch” between her and Jack
  - we know they are different people (presumably Jack is male)

Code Generation

- A translation into another language
  - Analogous to human translation

- Compilers for C, C++,…
  - produce assembly code (typically)
- (e.g. GUI) code generators
  - produce C, C++, Java,…

Summary: The Structure of a Compiler


program (stream of characters)
lexer
stream of tokens
parser
Abstract Syntax Tree (AST)
checker
AST with annotations (types, declarations)
code gen
maybe x86

Building a lexer

Recall: Lexical Analysis

- The input is just a sequence of characters. Example:

```
if (i == j)
        z = 0;
else
        z = 1;
```

- More accurately, the input is string:

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

- Goal: find lexemes and map them to **tokens**:
  1. **partition** input string into **substrings** (called **lexemes**), and
  2. **classify** them according to their **role** (role = **token**)

Continued

- Lexer input:

  `\tif(i==j)\n\t\tz = 0;\n\telse\n\t\tz = 1;`

- partitioned into these *lexemes*:

\t	if	(	i	==	j	)	\n\t\t	z	=	0	;	\n\t	else	\n\t\t	z	=	1	;

- mapped to a sequence of *tokens*

  IF, LPAR, ID("i"), EQUALS, ID("j") …

- Notes:
  - whitespace lexemes are dropped, not mapped to tokens
  - some tokens have **attributes**: the **lexeme** and/or **line number**
    - why do we need them?

How to build a lexer?

Writing the lexer

- Not by hand
  - tedious, repetitious, error-prone, non-maintainable
- Lexer generator!
  - once we have the generator, we'll only describe the lexemes and their tokens …
    - that is, provide language's lexical specification (the What)
  - … and generate code that performs the partitioning
    - generated code hides repeated code (the How)

Code generator: key benefit

- The lexer generator allows the programmer to focus on:
  - What the lexer should do,
  - rather than How it should be done.

- what: declarative programming
- how: imperative programming

Imperative lexer (in Java)

- Let's first build the lexer in Java, by hand:
  - to see how it is done, and where's the repetitious code that we want to hide
- A simple lexer will do. Only four tokens:

TOKEN	Lexeme
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	"=="
PLUS	"+"
TIMES	"*"

Imperative lexer

```
c=nextChar();
if (c == ‘= ‘) { c=nextChar(); if (c == ‘= ‘) {return EQUALS;}}
if (c == ‘+’ ) { return PLUS; }
if (c == ‘*’ ) { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

Maximal match rule

- What is the need for undoNextChar()?
  - it performs look-ahead, to determine whether the ID lexeme can be grown further

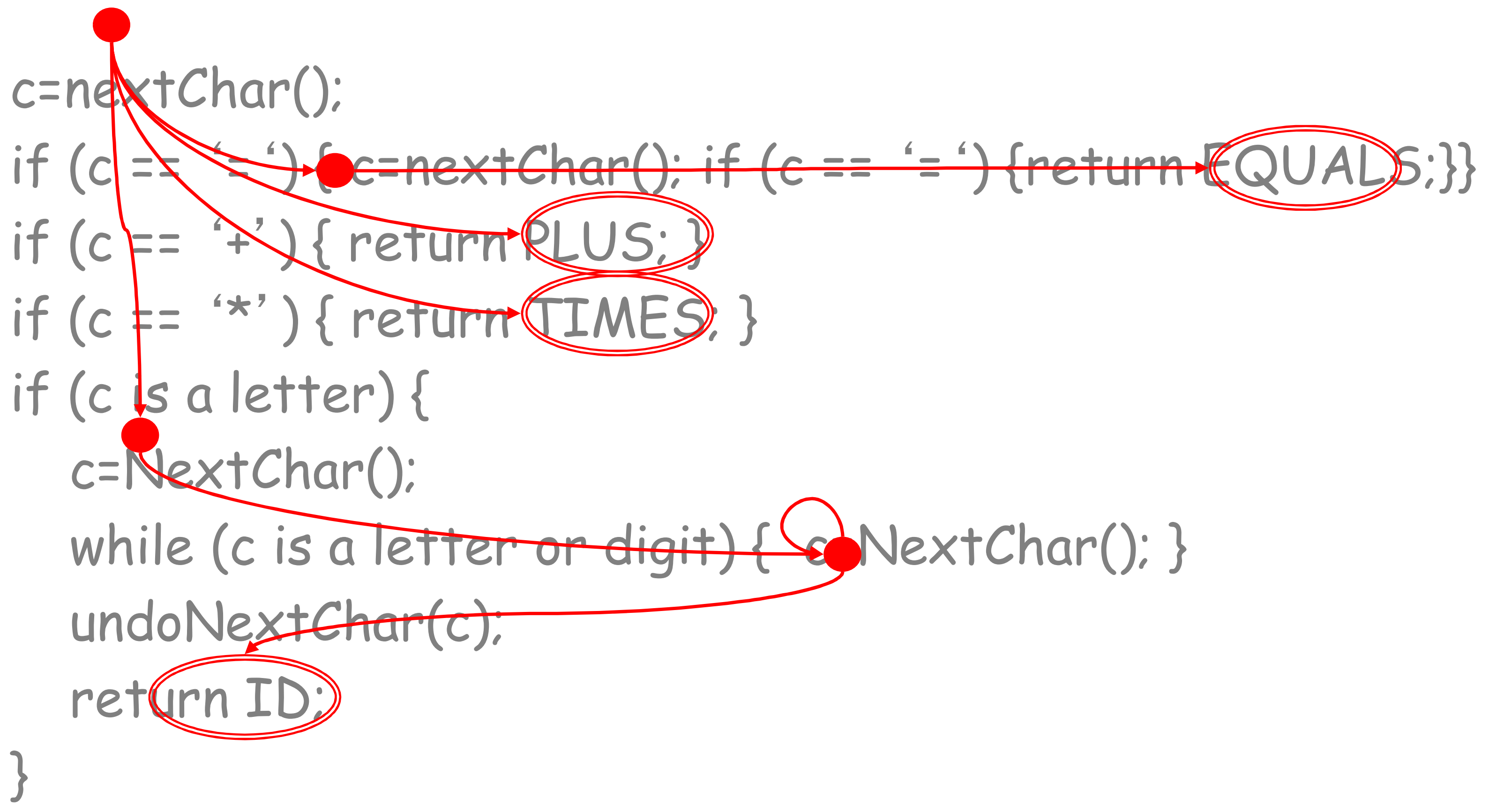
- This is an example of maximal match rule:
  - this rule followed by all lexers
  - **the rule:** the input character stream is partitioned into lexemes that are as large as possible
  - **Ex.:** in Java, “iffy” is not partitioned into “if” (the IF keyword) and “fy” (ID), but into “iffy” (ID)

Imperative Lexer: what vs. how

```
c=nextChar();
if (c == '= ') { c=nextChar(); if (c == '= ') {return EQUALS;}}
if (c == '+' ) { return PLUS; }
if (c == '*' ) { return TIMES; }
if (c is a letter) {
   c=NextChar();
   while (c is a letter or digit) {  c=NextChar(); }
   undoNextChar(c);
   return ID;
}
```

- little logic....

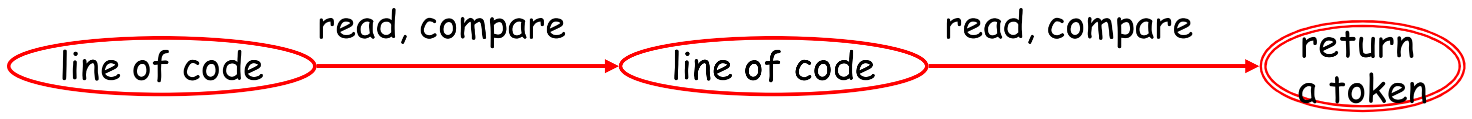
Separate out the what



```
c=nextChar();
if (c ==  ‘=’) { c=nextChar(); if (c ==  ‘=’) {return EQUALS;}}
if (c ==  ‘+’) { return PLUS; }
if (c ==  ‘*’ ) { return TIMES; }
if (c is a letter) {
    c=NextChar();
    while (c is a letter or digit) { c=NextChar(); }
    undoNextChar(c);
    return ID;
}
```

Separate out the what

- Is there a computational model that follows this behaviour?
  - yes, finite automata!



- The code actually follows this simple pattern:
  - read next character and compare it with some predetermined character
  - if there is a match, jump to a different line of code
  - repeat this until you return a token.

A declarative lexer

Part 1: declarative (the what)

- describe each token as a finite automaton (or, better, as a regular expression)
  - must be supplied for each lexeme, of course (it specifies the lexical properties of the input language)

Part 2: imperative (the how)

- connect these automata into a lexer automaton
  - common to all lexers (like a library)
  - responsible for the mechanics of scanning

the declarative lexer

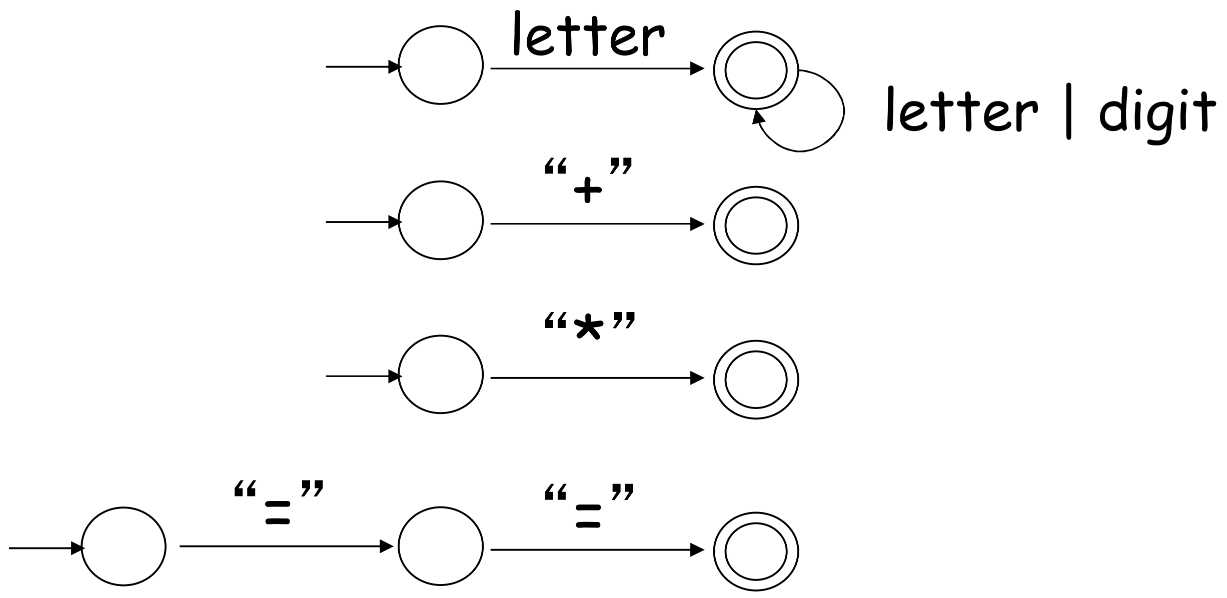
Part 1: specify a FA for each token

ID:

PLUS:

TIMES:

EQUALS:

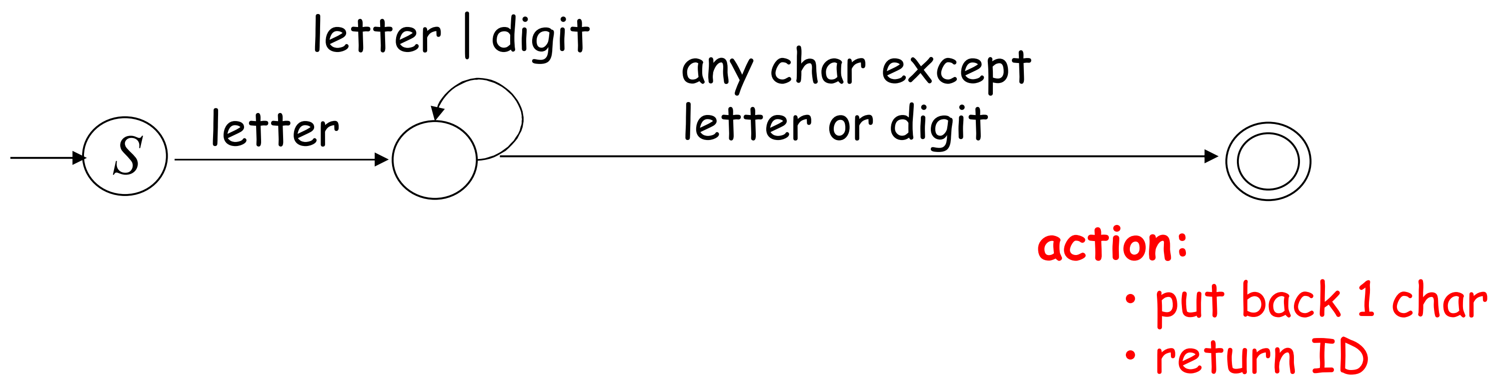


Part 2: add actions on acceptance of the FA

- the action can be one of
  - "put back one character" (if look-ahead is needed)
  - "return token XYZ"

Part 2: example of extending a FA

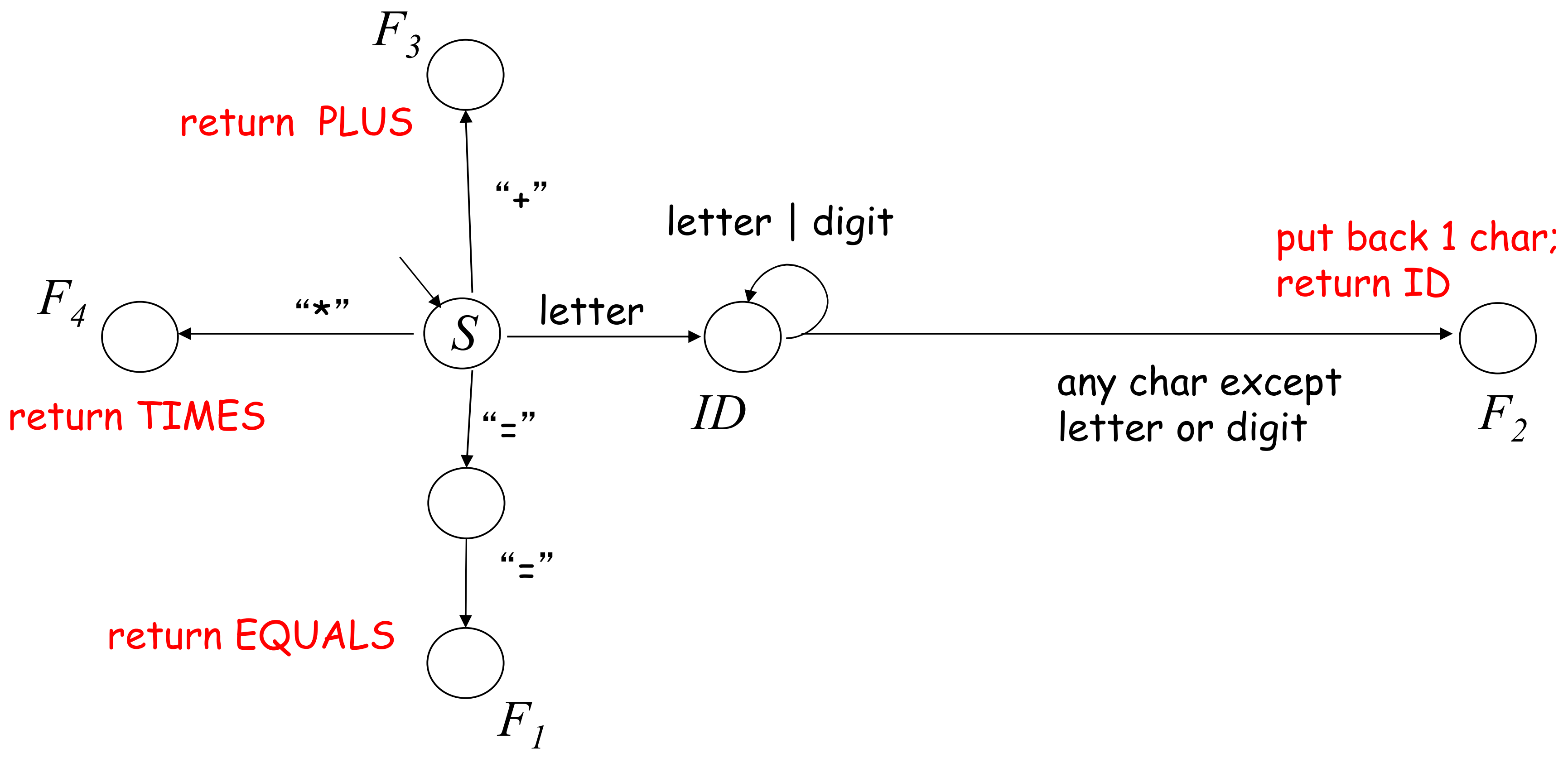
- The DFA recognizing identifiers is modified to:



- Look-ahead is added for lexemes of variable length
  - in our case, only ID needs lookahead
- A note on action "**return ID**"
  - resets the lexer back into start state S (recall that lexer is called by parser; each time, one token is returned)

Part 2: Combine the extended FA's

The algorithm: merge start states of the DFA's.

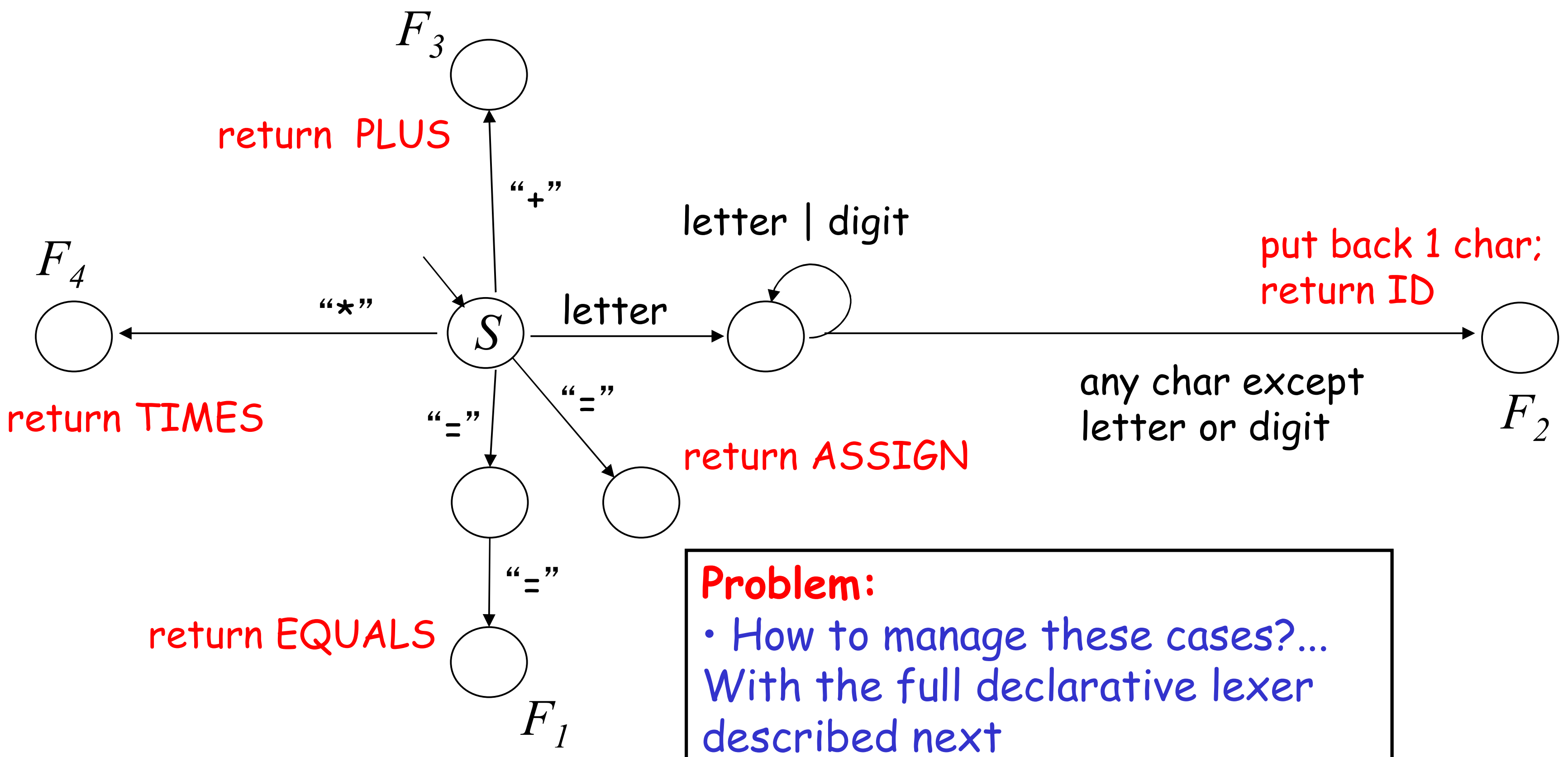


Towards a realistic scanner

- Consider a fifth token type, for the assignment operator

TOKEN	**Lexeme**
ID	a sequence of one or more letters or digits starting with a letter
EQUALS	“==“
ASSIGN	“=”
PLUS	“+”
TIMES	“*”

But above algorithm produces



Problem:

- How to manage these cases?... With the full declarative lexer described next

full declarative lexer

The algorithm

We can develop the lexer as follows:

1. Specify an automaton for each lexeme (as before)

Step 1 in our example

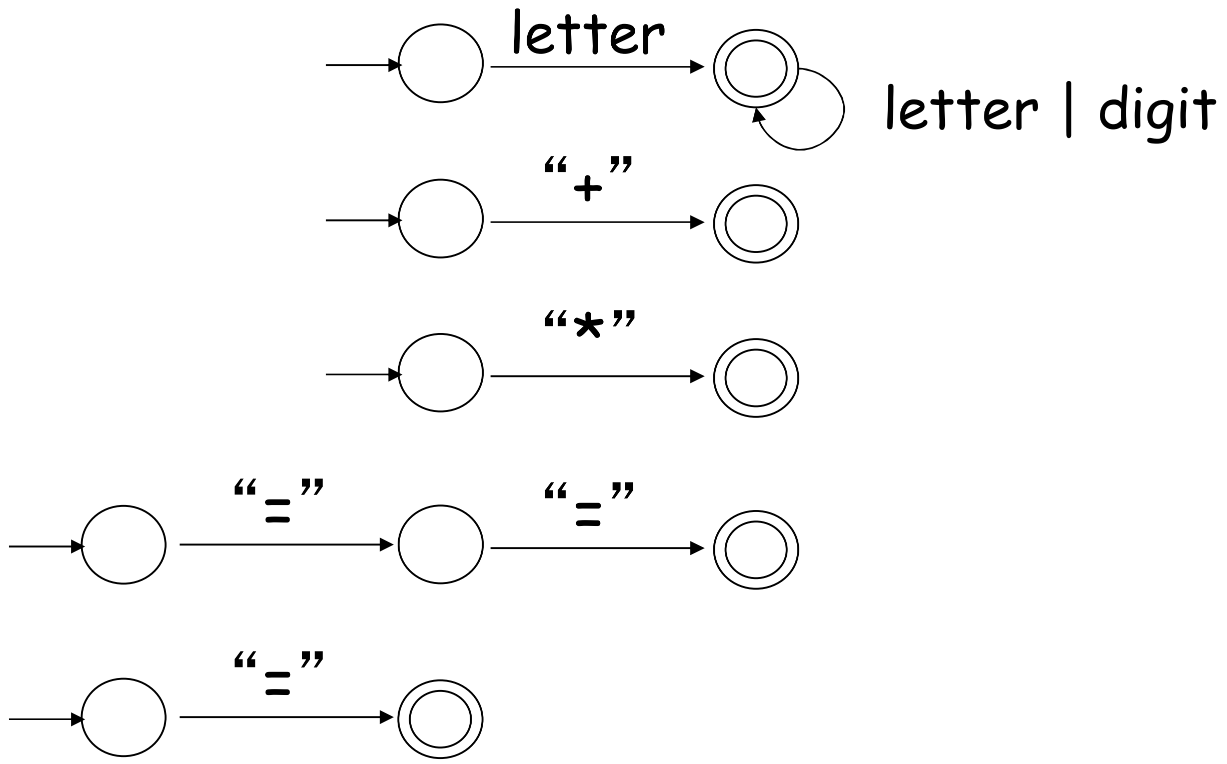
ID:

PLUS:

TIMES:

EQUALS:

ASSIGN:



The algorithm

We can develop the lexer as follows:

1. Specify an automaton for each lexeme (as before)
2. Merge them (start states) into the lexer automaton
3. If the obtained automaton is nondeterministic translate into a deterministic automaton
4. Consider the following rules (implementing maximal match):
   - whenever you reach a <u>final state</u>:
     - remember position in input (so that you can undo reads)
     - keep reading more characters, moving to other states
   - whenever you get <u>stuck</u> (cannot make a move on next char):
     - return to the last final state (i.e., undo the reads)
     - return the token associated with this final state

Notes

- Notice that reading past final state implements look-ahead
- This look-ahead is unbounded
  - can undo any number of characters

Practical concerns

- Ambiguity
  - **problem:** lexer may reach multiple final states at once
  - **ex.:** "if" matches both ID and IF (the keyword)
  - **solution:** prioritize tokens (IF wins over ID)
- Discarding whitespace
  - **solution:** final state for white-space lexeme is special: **don't return a token**, just jump to (common) start state
- Error inputs
  - **problem:** discard illegal lexemes and print an error message
  - **simple solution** (discard char by char): add a lexeme that matches any character, giving it lowest priority; it will match when no other will

We have a full declarative scanner

- imperative part,
  - The steps 2, 3, and 4 of the algorithm
- declarative part
  - Specify each token with an automaton

But how to specify the automata in practice?

regular expressions

Regular Expressions

- Automaton is a good "visual" aid
  - but is not suitable as a specification
    (its textual description is too clumsy)
- regular expressions are a suitable *specification*
  - a compact way to define a language that can be accepted by an automaton.
- used as the input to a lexer generator
  - define each token, and also
  - define white-space, comments, etc
    - these do not correspond to tokens,
      but must be recognized and ignored.

Example: identifier

- Lexical specification (in English):
  - a letter, followed by zero or more letters or digits.
- Lexical specification (as a regular expression):
  - [a-zA-Z]. ([a-zA-Z] | [0-9])*

|	means "or"
.	means "followed by"
*	means zero or more instances of
()	are used for grouping

Operands of a regular expression

- Operands are same as labels on the edges of an FA
  - single characters, or
  - the special empty string character ε
    (in ANTLR denoted by not writing any character!)

- sometimes we put the characters in quotes, e.g. *'a'* (required in ANTLR)
  - necessary when denoting | . *
- [a-z] (written 'a'..'z' in ANTLR) is a shorthand for
  - *a | b | c | … | z*
- [0-9] (written '0'..'9' in ANTLR) is a shorthand for
  - *0 | 1 | … | 9*

| . * operators have implicit precedence!

Regular Expression Operator	Analogous Arithmetic Operator	Precedence
|	plus	lowest
.	times	middle
*	exponentiation	highest

- Consider regular expressions:
  - [a-zA-Z] . [a-zA-Z] | [0-9]*
  - [a-zA-Z] . ([a-zA-Z] | [0-9])*

Example: Integer Literals

- An integer literal with an optional sign can be defined in English as:
  - "(nothing or + or -) followed by one or more digits"
- The corresponding regular expression is:
  - (+|-|ε).([0-9].[0-9]*)
- "." can also be omitted, e.g.
  - (+|-|ε) ([0-9] [0-9]*)
- A new convenient operator '+'
  - same precedence as '*'
  - [0-9].[0-9]* is the same as
  - [0-9]+ which means "one or more digits"

Lexer generators

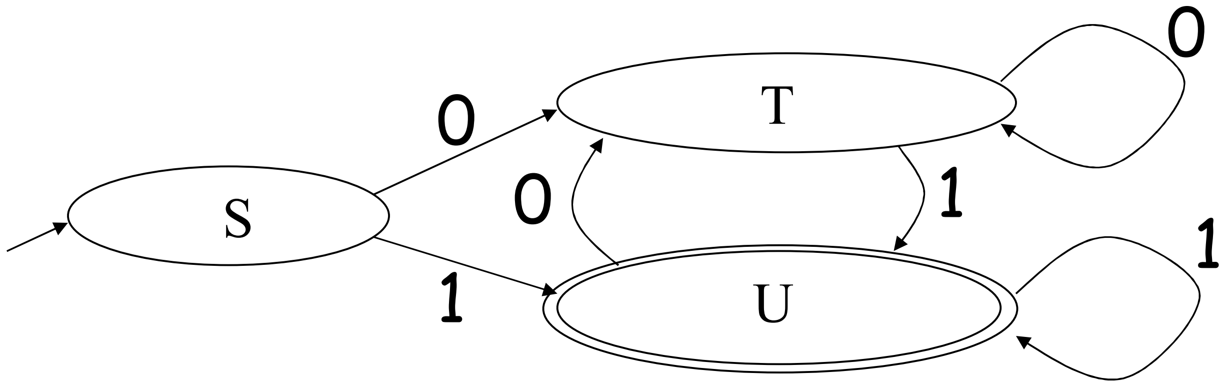
Lexer generators

- Receive in input the regular expressions describing the lexemes
- Generate code (C, C++, Java, …) that implements the full declarative lexer algorithm:
  - Translate the regular expressions into FA
  - Merge the FA in a unique automaton
  - Translate the merged automaton to a Determinisitic FA (more efficient to be simulated)
  - Produce the code that implements the "special" simulation of the DFA (lookahead for maximal match rule, priorities in case of multiple match, operations to be executed upon matching, and return to initial state)

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition $S_i \rightarrow^a S_k$ define T[i,a] = k
- DFA "execution"
  - If in state S_i and input a, read T[i,a] = k and skip to state S_k
  - Very efficient

Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

Example: flex (a fast lexer generator)

- **flex** generates lexers
- flex input:
  - file with regular expressions defining the tokens, and for each token some ***C*** intructions
- flex output:
  - a ***C*** file named **lex.yy.c** that contains the definition of a function **yylex()**
  - **yylex()** reads the input and finds the tokens
  - when a token is found the corresponding ***C*** instructions are executed

Example of flex input file

```
%{
#include <stdio.h>
%}
%%
[ \t\n]+      printf("white space, length %i\n",yyleng);
”*”           printf("times\n");    “/”           printf("div\n");
"+”           printf("plus\n");     "-”           printf("minus\n");
"(”           printf("left-par\n"); ")”           printf("right-par\n");
0|([1-9][0-9]*)                     printf("integer %s\n",yytext);
[a-zA-Z_][a-zA-Z0-9_]*              printf("identifier %s\n",yytext);
.                                   printf("illegal char %s\n",yytext);
%%
main(){ yylex(); }
```

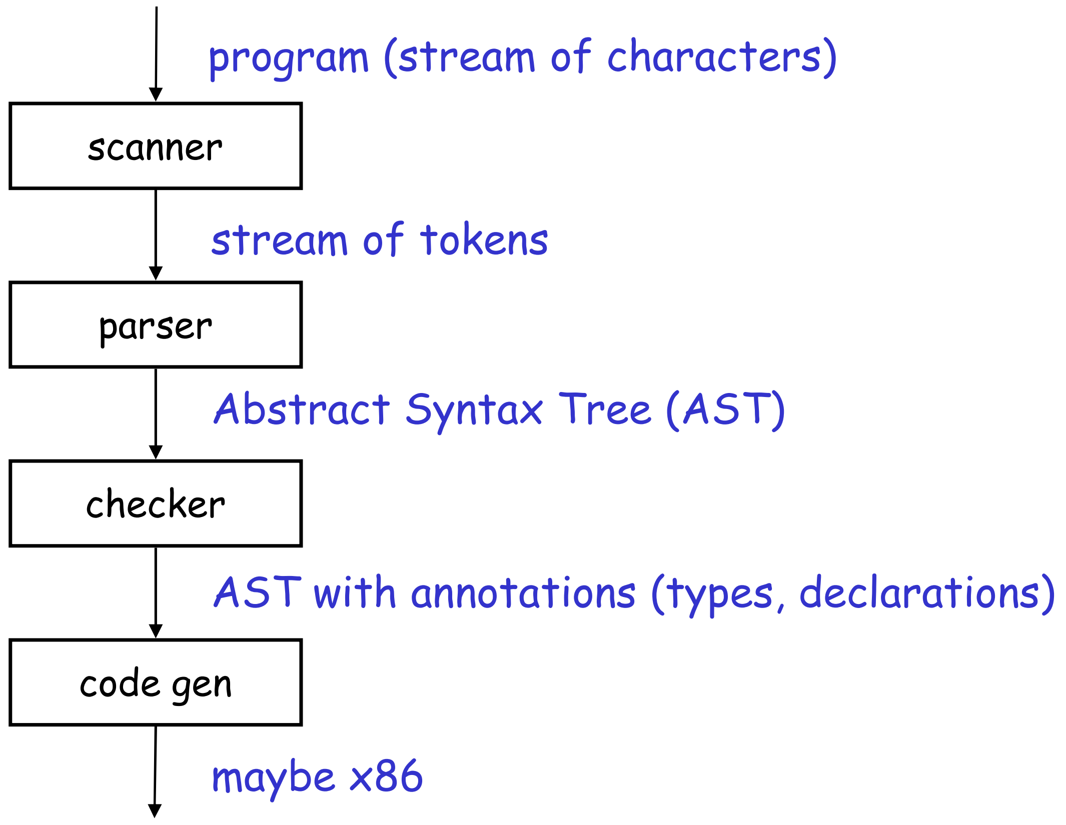
Building a parser

Overview

- What does a parser do, again?
  - its two tasks
  - parse tree vs. AST
- A hand-written parser
  - and why it gets hard to get it right

What does a parser do?

Recall: The Structure of a Compiler


program (stream of characters)
scanner
stream of tokens
parser
Abstract Syntax Tree (AST)
checker
AST with annotations (types, declarations)
code gen
maybe x86

Recall: Syntactic Analysis

- **Input:** sequence of tokens from scanner
- **Output:** abstract syntax tree
- Actually,
  - parser first considers the parse tree, i.e. the (non abstract) syntax tree
  - AST is then built by translating the parse tree
  - parse tree rarely built explicitly; only determined by, say, how parser pushes stuff to stack
  - our lectures first focus on constructing the parse tree; later we'll show the translation to AST.

Example

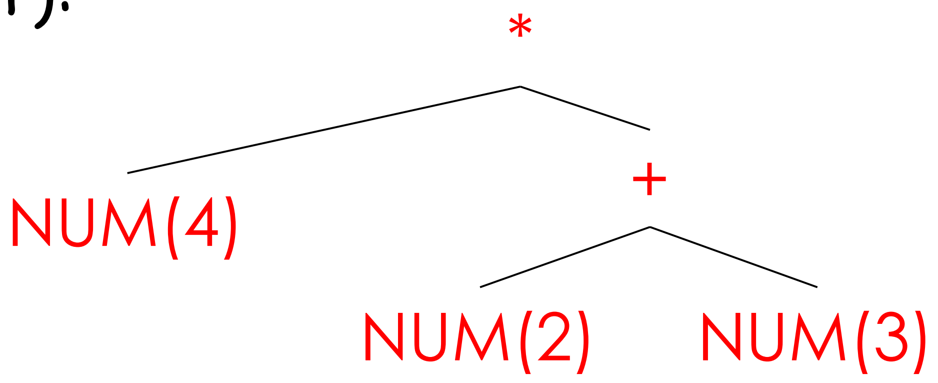
- Expression

  4*(2+3)

- Parser input

  NUM(4) TIMES LPAR NUM(2) PLUS NUM(3) RPAR

- Parser output (AST):



Parse tree for the example

$$\begin{tikzcd}
 & & & \text{EXPR} \arrow[dddlll] \arrow[dddll] \arrow[dr] & & & \\
 & & & & \text{EXPR} \arrow[ddll] \arrow[d] \arrow[ddrr] & & \\
 & & & & \text{EXPR} \arrow[dl] \arrow[d] \arrow[dr] & & \\
\text{NUM(4)} & \text{TIMES} & \text{LPAR} & \text{NUM(2)} & \text{PLUS} & \text{NUM(3)} & \text{RPAR}
\end{tikzcd}$$

leaves are tokens

Another example

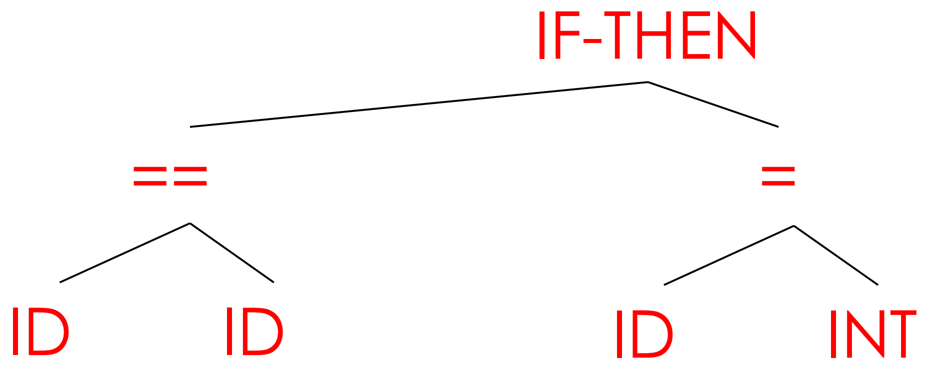
- Other expression

  if (x == y) { a=1; }

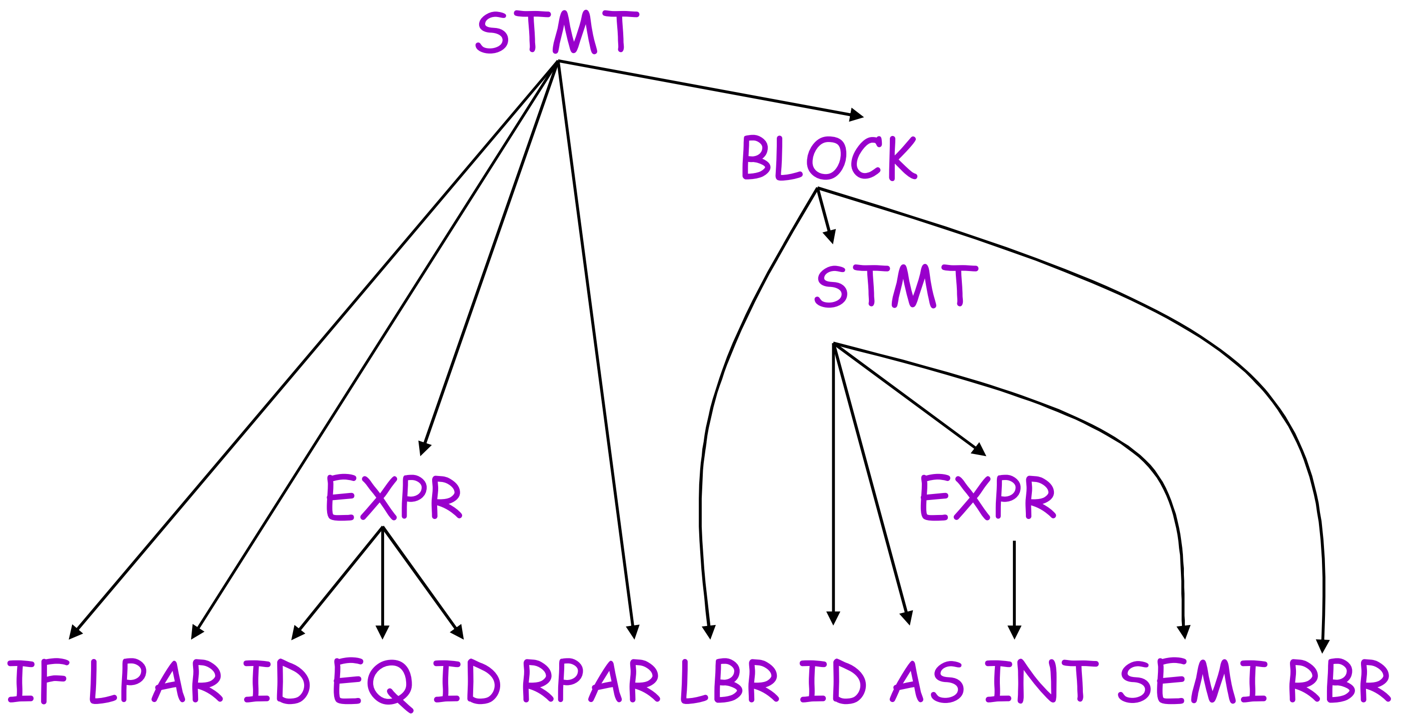
- Parser input

  IF LPAR ID EQ ID RPAR LBR ID AS INT SEMI RBR

- Parser output (AST):



Parse tree for the example



leaves are tokens

Parse tree vs. abstract syntax tree

- Parse tree
  - contains all tokens, including those that parser needs "only" to discover
    - intended nesting: parentheses, curly braces
    - statement termination: semicolons
  - technically, parse tree shows **concrete** syntax
- Abstract syntax tree (AST)
  - abstracts away artifacts of parsing, by flattening tree hierarchies, dropping tokens, etc.
  - technically, AST shows **abstract** syntax

Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	AST, built from parse tree

Summary

- Parser performs two tasks:
  - Syntax checking
    - a program with a syntax error is rejected and information about error given
  - Parse tree construction
    - usually implicit (needed for syntax checking)
    - exploited to build the AST

How to build a parser?

Writing the parser

- Can do it all by hand, of course
  - ok for small languages, but hard for real programming languages
- Just like with the scanner, it is possible to use a parser generator
  - one concisely describes the syntactic structure
    - that is, how expressions, statements, definitions look like
  - and the generator produces a working parser
- Let's start with a hand-written parser
  - to see why we want a parser generator

First example: balanced parens

- Our problem: check the syntax
  - are parentheses in input string balanced?
- The simple language
  - parenthesized number literals
  - Ex.: 3, (4), ((1)), (((2))), etc
- Before we look at the parser
  - why aren't finite automata sufficient for this task?

Parser code preliminaries

- Let TOKEN be an enumeration type of tokens:
  - NUM, LPAR, RPAR, PLUS, MINUS,TIMES, DIV

- Let the global in[] be the input string of tokens

- Let the global next be an index in the token string (initially is 0)

Parsers use stack to implement infinite state

Balanced parentheses parser:

```
void Parse() {
   nextToken = in[next++];
   if (nextToken == NUM) return;

   if (nextToken != LPAR)  print("syntax error");
   Parse();
   if (in[next++] != RPAR) print("syntax error");
}
```

Where's the parse tree constructed?

- In this parser, the parse is *given* by the call tree:
- For the input string (((1))) :

$$\begin{tikzcd}
 & \text{Parse()} \arrow[dl] \arrow[d] \arrow[dr] & \\
(& \text{Parse()} \arrow[dl] \arrow[d] \arrow[dr] &) \\
(& \text{Parse()} \arrow[dl] \arrow[d] \arrow[dr] &) \\
(& \text{Parse()} \arrow[d] &) \\
 & 1 &
\end{tikzcd}$$

Second example: subtraction expressions

The language of this example:

1, 1-2, 1-2-3, (1-2)-3, (2-(3-4)), etc

```
void Parse() {
    if ((nextToken = in[next++]) == NUM) {
            if (in[next] == MINUS)  { next++; Parse(); }
    } else if (nextToken == LPAR) {
            Parse();
            if (in[next++] != RPAR) print("syntax error");
    } else print("syntax error");
}
```

Second example: subtraction expressions

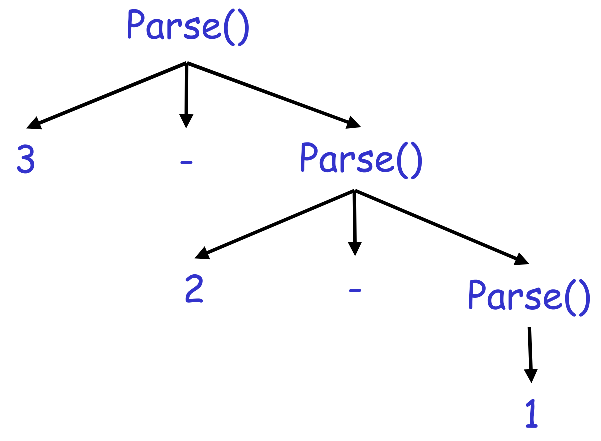
The language of this example: (fixed!!)

1, 1-2, 1-2-3, (1-2)-3, (2-(3-4)), etc

```
void Parse() {
    if ((nextToken = in[next++]) == NUM) {
            if (in[next] == MINUS)  { next++; Parse(); }
    } else if (nextToken == LPAR) {
            Parse();
            if (in[next++] != RPAR) print("syntax error");
            if (in[next] == MINUS)  { next++; Parse(); }
    } else print("syntax error");
}
```

Subtraction expressions continued

- Observations:
  - a more complex language
    - hence, harder to see how the parser works (and if it works correctly at all)
  - the parse tree is actually not really what we want
    - consider input 3-2-1
    - what's undesirable about this parse tree's structure?



We need a clean syntactic description

- Just like with the scanner, writing the parser by hand is painful and error-prone
  - consider adding +, *, / to the last example!

- So, let's separate the what and the how
  - what: the syntactic structure, described with a **context-free grammar**
  - how: the parser, built from the grammar, which reads the input and produces the parse tree

Idea…

- We can describe the syntactic structure by using context-free grammars!
- What’s next?
  - Grammars
  - Derivations

Grammars

- Programming language constructs have recursive structure.
  - which is why our hand-written parser had this structure, too

- Example: an expression is either:
    - number, or
    - variable, or
    - expression + expression, or
    - expression - expression, or
    - (expression), or
    - …

Context-free grammars (CFG)

- a natural notation for this recursive structure
- grammar for our balanced parens expressions:

  BalancedExpression $\rightarrow$ **a** | **(** BalancedExpression **)**
- describes (generates) strings of symbols:
  - a, (a), ((a)), (((a))), …
- like regular expressions but can refer to
  - other expressions (here, BalancedExpression)
  - and do this recursively (thus, it is "non-finite state")

Example: arithmetic expressions

- Simple arithmetic expressions:
  $E \rightarrow n \mid id \mid (E) \mid E + E \mid E * E$

- Some strings of this language:
  - id
  - n
  - (n)
  - n + id
  - id * (id + id)

How do derivations help us in parsing?

- A program (a string of tokens) has no syntax error if it can be derived from the grammar.
  - grammars, however, define how to derive some (any) string, not how to check if a given string is derivable
- So how to do parsing?
  - a naïve solution: derive all possible strings and check if your program is among them
  - not as bad as it sounds: there are parsers that do this. Coming soon.

Example

A fragment of a simple language:

```
STMT  →  while (  EXPR ) STMT
       | id ( EXPR ) ;

EXPR  → EXPR + EXPR
       | EXPR - EXPR
       | EXPR <  EXPR
       | ( EXPR )
       | id
```