

CSE 131A
“Compiler Construction I”

Lecture 3

Overview of language translation

Lexical scanning

Today's lecture

- A few Onyx details
- Lexical scanning

Announcements

- A0 due on Weds
- Office hours schedule **next** week
 - Monday 1/22 are CANCELLED
 - Weds 1/24 : Moved to Thursday 1/25, 5 to 6pm

Return

- All of these return the same result

```
let $a := 2
  let $b := 3
  return $a*$b
```

```
let $a := 2
  return(
    let $b := 3
    return $a*$b
  )
```

```
let $a := 2
  return
    let $b := 3
    return $a*$b
```

DOM behavior

- When adding a child to an element node, duplicates are ignored
- The Onyx XML library inherits this behavior from the W3C DOM
- Only the most recently added child is represented in the output

```
let $a := attenv()
```

```
let $e := enode("S", $a), $c := enode("foo", $a)
```

```
let $e := addChildNode($e, $c)
```

```
let $t := tnode("T", $a, "e"), $c := addChildNode($c, $t)
```

```
let $e := addChildNode($e, $c)
```

```
return $e
```

```
<S>  
  <foo>  
    <T>e</T>  
  </foo>  
</S>
```

DOM behavior

- Use a different form of the constructor instead of side-effect producing operations (not currently implemented in Onyx)

```
let $a := attrenv()  
let $t := tnode("T", $a, "e"),  
    $foo := enode("foo", $a, $t),  
    $$ := enode("S", $a, $foo),  
return $$
```

```
for $i in (3,5)  
    return $i * $i
```

```
<S>  
  <foo>  
    <T>e</T>  
  </foo>  
</S>
```

Next

- Overview of compilers
- Introduction to lexical analysis

How do we implement a programming language?

- Translate the source code into commands for an on-line data base server

```
let $d := document("stats.xml"),
    $teams := getDescendants( $d, "TEAM_CITY" )
for $t in $teams
    return $t
```

- For example, the Onyx program shown would result in commands such as:
 - Open the OXML data base “stats.xml”
 - Call the user-defined function getDescendants to extract all elements with the tag “TEAM_CITY”
 - Return each “TEAM_CITY” element

Modularized translation

- In the course you will build a modular translator
- The modules are: lexer, parser, semantic checker/interpreter
- This modular approach is a standard software engineering technique to simplify code development and maintenance
- However, it is possible to combine parsing, semantic analysis, and code generation in one “module” using syntax-directed translation... we’ll cover this later on

Steps of Translation

- *Lexical analysis*: group the input characters into *tokens*

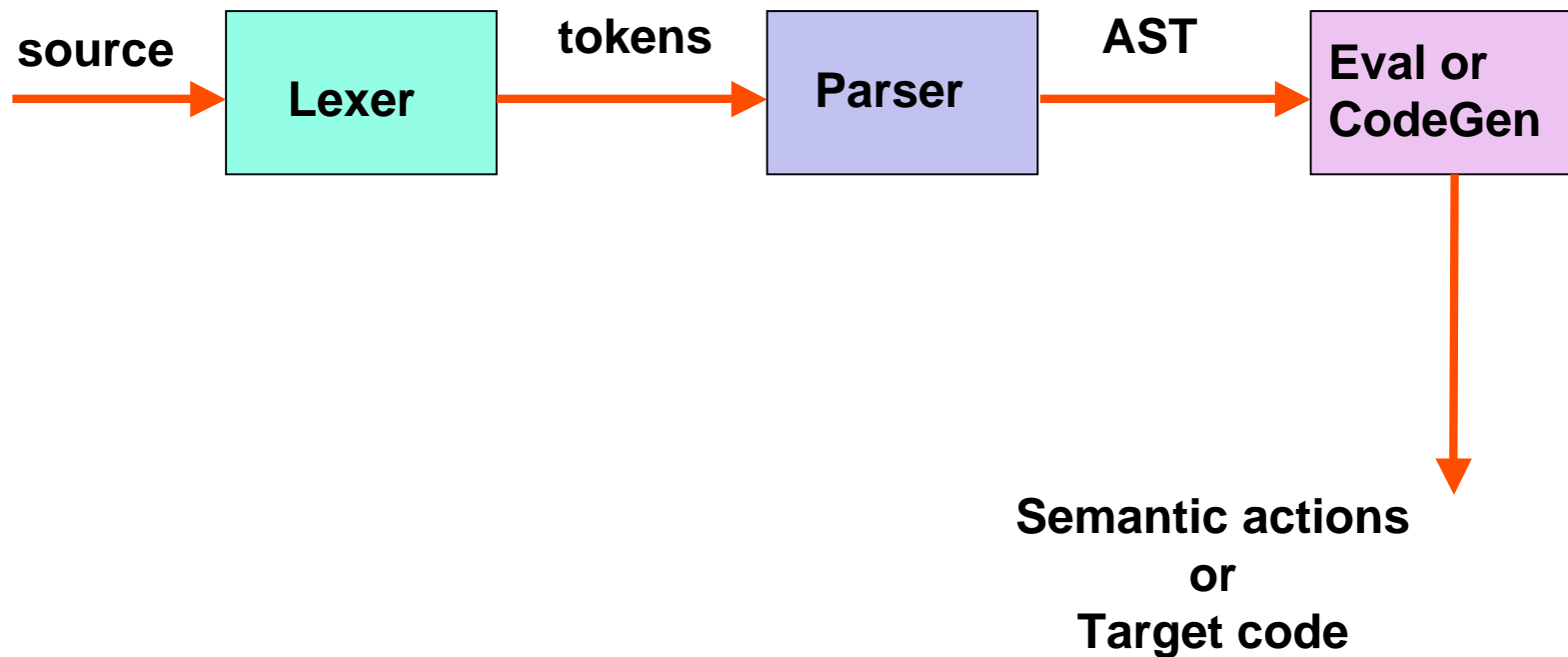
for \$p in 1 to 7

- *Syntactic analysis (parsing)*: determine if the grouping of the tokens is consistent with the language definition, and if so, what the structure of the input is. Generate an abstract syntax tree (AST)

for tokenVariable **in** ExprSingle (, tokenVariable in ExprSingle)*
return ExprSingle

- *Semantic analysis*: analyze and check semantic (e.g. type-related) properties of the input
- *Interpretation/code generation*: using the AST, carry out the operation (or produce target code) that does what the input program requires

Architecture of a compiler or interpreter



The practical side of the course

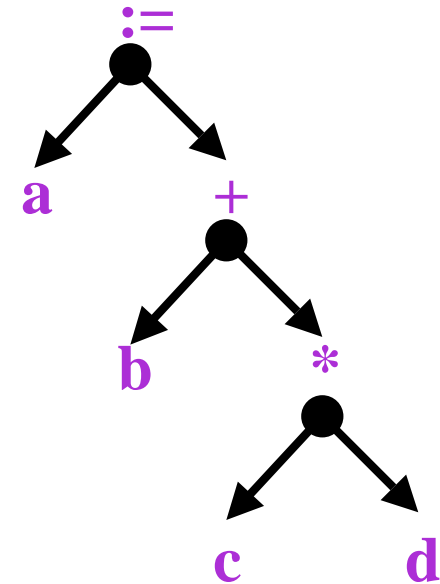
- We'll build an Onyx interpreter in 3 steps.
- *Lexical analysis* (Assignment #1)
 - Scanning and token generation
- *Parsing* (Assignment #2)
 - Syntax checking, error recovery
- *Semantic analysis and some code generation* (#3)
 - Type checking
 - Target is the Onyx Document Object Model *ODOM*
 - Optional code drop

Lexical Analysis

- The purpose of lexical analysis is to locate and identify *tokens* in an input text
 - A token corresponds to an important and meaningful piece of the input
 - In a compiler context, information about the recognized tokens is in turn input to the parser and subsequent code generation phases
 - Information about location of tokens can be used in error reporting
- Lexical analysis techniques are broadly applicable
 - Query processing, shell/Perl/Python scripting, report generation, simple command language interpretation

Parsing

- Take as input tokens from the lexical analysis phase
- Organize the tokens into a hierarchical representation according to recursive syntactic rules
- A *context-free grammar* defines the rules, so syntactic structure can be recognized by pushdown automata
- Example: a simple typical assignment expression



$a := b + c * d$

Semantic checking and code generation

- Take as input the structure produced in the parsing phase
- Using semantic rules for the language..
 - check that the structure “makes sense,” e.g. doesn’t violate type rules
 - if so, output or execute code that implements the required semantics
- Semantic rules are typically attached to context-free grammar productions, but may exceed the expressive power of context-free languages

Lexical analysis in context

- A lexical analyzer reads input text and produces a sequence of token representations that a parser uses for syntax analysis
- Usually the lexical analyzer is implemented as a subroutine or coroutine of the parser
- The parser emits a “get_next_token” message to the lexer, which reads input characters until it can recognize the next token, and returns to the parser a representation of the token together with some other information

Tokens, patterns, and lexemes

- A *token* corresponds to a set of character strings that are described by a rule or *pattern*
- The pattern is said to *match* each string in the set `[0-9]+`
- A *lexeme* is a particular character string in the input matching the pattern for some token `555 376 0`

What patterns define tokens?

- A token corresponds to strings that match a particular pattern of characters in the input text
- Which patterns you want to recognize depend on the application, but...
- ... in a lexical analyzer, the strings for a token are restricted to be a *regular language*
- Contrast this with parsers, which can recognize *context-free languages*

Why separate the lexer and parser?

- As you know from formal language theory, the context-free languages contain the regular languages...
- ... so a parser could recognize all the patterns that a lexer recognizes.
- Then why have a separate lexer?
- The answer derives from basic principles of software system design

Modularity in compiler design

- **Simplicity:** a lexer can easily filter e.g. whitespace and comments from the input; this significantly simplifies the parser's job
- **Efficiency:** scanning text input can be time consuming, but a specialized lexer can be optimized for the task
- **Portability and maintainability:** the lexer is insensitive to the target hardware and can be independently modified if needed

Creating a lexical analyzer

- A lexical analyzer or lexer is a program that ...
 - recognizes regular language token patterns found in input text
 - takes actions according to those patterns
- We require an precise specification for the token patterns and associated actions
- We build an analyzer that recognizes the language and carries out the specified actions
- Rather than building our own scanner from scratch, we'll use a lexical analyzer generator such as **Jflex**

Why use a lexical analyzer generator?

- Hand construction of an analyzer is time consuming and error prone, especially if the language is undergoing revision
- There are good algorithms and tools to generate lexical analyzers that encapsulate expert knowledge
- We are free to focus on more challenging concerns
 - Error recovery
 - Semantic analysis
 - Code generation and optimization
- We can learn by example how to construct large software systems by making use of available tools

Regular expressions

- Lexer token patterns are specified with *regular expressions*
- Regular expressions are a notation for defining regular languages
- Many languages and systems use regular expressions for pattern matching and processing; they are a powerful computational tool
 - Unix Shell languages, Unix tools ed/sed/awk/grep, vi, emacs, Perl, Python, Java 1.5, ...
- Regular expression notations vary somewhat; we'll concentrate on what **Jflex** uses

Toward Regular Expressions

- Regular expressions represent operations on regular languages
- These operations can “build up” and define arbitrary regular languages
- Let’s review the basics of regular languages and operations on them

Alphabets

- An *alphabet* or *character set* Σ is a finite set of symbols or letters
- Examples:
 - The *binary alphabet* is the set $\Sigma = \{0, 1\}$
 - We will use 16-bit Unicode 2.0 chars for our Onyx alphabet
 - 7-bit ASCII is also a common character class

Strings

- A *string* over an alphabet Σ is a finite-length sequence of symbols drawn from Σ
- Example:
A, AB, FOOBARBAZ are 3 strings over the alphabet of uppercase Roman letters
- The set of *all* strings over Σ is written Σ^*
* is the Kleene closure operator: all possible concatenations of 0 or more items drawn from the set
- Example:
The strings from the binary alphabet are $\{0,1\}^* = \varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, \dots$

String length and the empty string

- Synonyms for a string are *sentence* and *word*
- The length of a string s is the number of alphabet symbols in s and is written as $|s|$
 $|\text{supercalifragilisticexpialidocius}| = 33$
- The empty string ϵ has length 0

Languages

- A *language* is a set of strings over a fixed alphabet
 - If Σ is in the alphabet, a language L over Σ is a subset of Σ^* , i.e. $L \subseteq \Sigma^*$
 - We sometimes call the strings in L *words* or *sentences* in the language
- Examples
 - Correct written forms of natural languages
 - Þá er ég kominn á nýjan stað
 - Наши дети 富临海鲜
 - Syntactically well formed Java programs
 - Legal C++ integer literals

Basic Operations on Languages

- Let $L \subseteq S^*$, $M \subseteq S^*$ be languages over S
- **Union:** $L \cup M$ is the language whose words are words in either L or M
- **Concatenation:** LM is the language whose words are a concatenation of a word from L and a word from M
- **Kleene star:** L^* is the language whose words are concatenations of 0 or more words from L
- **Complement** (with respect to S) : $\neg L$ is the language whose words are in S^* but not in L
- **Intersection:** $L \cap M$ is the language whose words are words in both L and M ; equivalent to $\neg(\neg L \cup \neg M)$

Regular Languages

- Regular languages $RL(\Sigma)$ over an alphabet Σ can be defined inductively
- For each symbol $s \in \Sigma$, $\{s\}$ is in $RL(\Sigma)$
- We build up the languages using the following rules
- If languages L, M are in $RL(\Sigma)$, then so are

$L \cup M$

Union

LM

Concatenation

L^*

Kleene closure

$\neg L$

Complement

$L \cap M$

Intersection

- We can construct any RL over Σ using a finite application of Union, concatenation, Kleene closure to Σ

Regular Expressions

- Regular expressions are notations for defining regular languages
- Operators in regular expressions refer to the essential operations on regular languages, plus some others included for convenience
- We'll look at regular expression notation in JFlex
- In the following, the alphabet Σ will be the set of Unicode characters;
E, E1, E2 will be arbitrary regular expressions

Characters and character sets

- Any Unicode character except a regular expression metacharacter denotes the RL consisting of just that character: `a`, `X`, `9`, `%`, `\u03a4`
 - Metacharacters are: `| () [] { } < > . $ ^ \ * + ? / " ~ !`
 - As simple characters, these must be backslash-escaped
- Any sequence of characters in brackets `[]` denotes a character set, i.e. the RL whose words are just those individual characters: `[a]`, `[%\$\&]`, `[0123456789]`
- Character ranges are allowed: `[0-9]`, `[a-zA-Z]`
 - `[0-9]` is equivalent to `[0123456789]`
 - `[a-zA-Z]` is equivalent to `[abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]`

Characters and character sets, cont'd

- A character set beginning with `[^` indicates the complement of the set: `[^abc]` is the RL whose words are all the Unicode characters except `a,b,c`
- `[]` denotes the empty language; `[^]` denotes Σ , i.e. the set of all one-character Unicode strings
- Predefined character sets: `[:digit:]`, `[:letter:]`, etc.
 - `[:digit:]` is equivalent to `[0-9]` is equivalent to `[0123456789]`
 - `[:letter:]` includes all the “letter” Unicode characters (there are more than 40,000 of these)

Union, concatenation, Kleene star

- $E1 \mid E2$: the RL that is the union of the RLs $E1$ and $E2$
- $E1 E2$: the RL that is the concatenation of the RLs $E1$ and $E2$
- E^* : RL that is the Kleene closure of the RL E
- Character concatenation: a string enclosed in double quotes “ ” denotes the RL consisting of the one word equal to the value of the string

Iteration, option

- E^+ : the RL whose words consist of concatenations of one or more words from the RL denoted by E
 - Equivalent to $E E^*$
 $a^+ \rightarrow a, aa, aaa, \dots$
- $E?$: the RL whose words consist of the RL denoted by E , together with the empty string
 - Equivalent to $E \mid \epsilon$
 $a? \rightarrow a, \epsilon$

Some examples

- $(a \mid b)^+$: the set of strings containing 1 or more occurrences of a 's and b 's
- $a \mid (a^*b)^+$
strings that are the single character 'a' strings and strings that can be formed by concatenating one or more strings each of which starts with 0 or more a 's followed by a single b :
{ a, b, bb, bbb, ..., ab, abab, bab, aabaab, . }

Examples to work at home

- For each of the following regular expressions, describe in words the RL it denotes:
- $0(0|1)^*0$
- $((""|0)1^*)^*$
- $0^*10^*10^*10^*$

Yet more examples

- Write a regular expression that denotes each of these languages:
- All Unicode strings that contain the 5 lowercase vowels a,e,i,o,u in order
- All strings of 0's and 1's containing an even number of 0's
- All strings of 0's and 1's containing an odd number of 1's
- All strings of 0's and 1's containing an even number of 0's and an odd number of 1's

Complement, up-to, enumeration

- $!E$: the complement of the RL denoted by E
- $\sim E$ “up-to:” the RL whose words consist of the concatenation of
 - a string that contains no word in E ...
 - followed by a word in the RL denoted by E
 - i.e. any string QE , such that no prefix of Q contains E
 $\sim abc \rightarrow$ helloabc but not abcabc or xabcabc
 - equivalent to $!([\wedge]^* E [\wedge]^* | "") E$

Enumeration, grouping

- If n is a positive integer, then $E\{n\}$ denotes the RL whose words consist of the concatenation of n words from the RL denoted by E
- If n, m are positive integers, $m \geq n$, then $E\{n,m\}$ denotes the RL whose words consist of the concatenation of at least n but not more than m words from the RL denoted by E
- (E) denotes the same RL denoted by E

Precedence

- Regular expression operators have this order of precedence, from highest to lowest:
 - Unary postfix: $*$, $+$, $?$, $\{n\}$, $\{n,m\}$
 - Unary prefix: $!$, \sim
 - Concatenation
 - Union: $|$
- Precedence can be overridden with use of parentheses. When in doubt, parenthesize!

Choosing Tokens

- If the lexer can recognize only regular languages, what regular languages should it recognize?
- Put it another way: What should the tokens of the input language be?
- This depends on what the input language is: tokens should be meaningful units of the language, that can be specified by regular expressions, and that will be useful to the parser

Meaningful units as tokens

- It makes sense to tokenize this Onyx expression as shown:

for \$p in document ("stats.xml")

- Each of the underlined lexemes is a meaningful unit in the language
- It would not be useful to tokenize the expression like this, for example:

for \$p in docu ment ("stats.xml")

Typical kinds of tokens

keywords	const for int
operators	+ - * /
identifiers	panelHeight
numeric constants	2.71828
literal strings	“hello world”
punctuation	() [] ;
comments	/* don't do that! */

Tokens and Programming Language Design

- We can design a language so that ...
 - a regular-expression driven lexical analyzer can efficiently tokenize input ...
 - ... for a context-free-grammar driven parser
- Most modern programming languages are designed with this in mind
- Older languages were not, and it can be very hard to build compilers for them

Whitespace and token delimiters

- In FORTRAN, white space is irrelevant
- A FORTRAN **do** loop may begin
`do i = 1, 25`
- But that is equivalent to
`doi=1,25`
- But the following is a legal FORTRAN assignment statement:
`doi=1`
- A lexer needs to look ahead to the comma to know how to tokenize this

White space in Onyx

- Most languages use white space as a way to delimit tokens; this makes lexical analysis easier
- In Onyx, white space is important in some contexts
- Onyx variable names can contain the - character which is also the minus operator; the difference depends on white space context
 - Part of variable: `$x-1 + 1`
 - Minus operator: `$x^-1 + 1`
- These contexts are easily definable in terms of regular languages, but OXML whitespace is more difficult
- We don't have OXML literals in Onyx

Reserved words

- Some languages reserve certain identifiers to use as keywords; these identifiers cannot be used otherwise
 - Java reserves **class**, **if**, **void**, and many others
 - Thus, the following expression is not legal in Java:
if = 7
- If a language has no reserved words, keywords can only be determined by their context
- If that context is not definable in terms of a regular language, lexical analysis cannot determine keywords, and the parser's job gets harder
- For example, this statement is legal in the PL/I language:
if then then then = else; else else = then

Reserved words in XQuery (W3C)

- The XQuery language specification has been revised several times
- With each revision, the number of reserved words in the language has been reduced
- The stated goal is to have *no* reserved words
- Rationale: XQuery permits embedded XML, and XML has no reserved identifiers; any QName can be a tag name
- This goal creates some interesting issues in compiler design

Comments

- A comment in a programming language is of no use to the parser; the lexer should just recognize comments in the input and discard them
- But this means that comments must be definable as a regular language
- This is why C-style comment syntax is defined the way it is: starting with “*/**” through the *next* “**/*”, not the *matching* “**/*”
- Which leads to syntax errors like this in C/C++/Java:

```
/* Comment out this dangerous code!  
    badFunction(3); /* dangerous */  
*/
```

- What is a regular expression definition of C-style comment?

Nesting comments

- What if we wanted to add nested comments?

```
/* Comment out this funny code!
```

```
    FunnyFun(x,y) /* humorous */
```

```
*/
```

- Regular languages can't handle “parenthesis matching”
- Context free languages can
- So either
 - the parser has to handle comments, making it much more complicated
 - or the lexer has to be more powerful than just a regular expression matcher (Jflex “states”)
- Onyx supports non-nested comments only

Recognizing tokens

- So far we've talked about how to specify tokens
 - Tokens are regular languages...
 - ... specified with regular expressions
- We next consider how to recognize them in a text stream and respond with actions
- We'll use a lexer generated automatically by JFlex