Final Review Sheet

Introduction

The final exam will be given 8:30 to 10:20 on Monday, December 16 in our regular classroom. You will have 1 hour and 50 minutes for the exam.

The exam will be closed book, no notes, no calculators.

The exam is based on the lectures and the homework. The questions will be similar to the questions in the homework and midterm exam with several small programming problems too. The exam will cover Java and language scanning and parsing. The final exam will not include any Scheme questions.

Compiler intro

Interpreter. Execution interleaved with program analysis.

Compiler. Translate program to semantically equivalent program in some other language like hardware instructions, bytecodes (virtual machine instructions), or even another high level language (eg, bison input file to C code). Tradeoffs among the various places that the overhead time can best be spent. Java compiler (javac, jikes) compile Java to bytecodes. MS .NET compiles to Microsoft intermediate language (MSIL).

As hardware gets faster and more complex (more parallelism) compilers are getting smarter and also generate code at a higher level. The distinction between compile time and run time is blurring - some “compilation” is taking place at run time when the run time executive recognizes that some sections of existing code are executing frequently and can benefit from further compilation (to a language closer to the hardware).

Compiler front end does source analysis, back end does code synthesis. Analysis is further split into scanner and parser primarily.

Scanner, Regular Expressions, Deterministic Finite Automata

Scanner converts character stream into tokens that are meaningful to the parser. Removes noise like comments, whitespace, invalid characters. Each parser-significant item in the input is represented by a token, agreed on by the scanner and the using module (eg, parser). Some tokens have attributes. For example an identifier token will have a string attribute containing the name of the identifier, a numeric constant token will have a numeric attribute containing the actual value.

Scanners generally based on using regular expressions to define and implement capabilities. Typical tokens represent the operators, punctuation, keywords, identifiers, and literals (constants) in a programming language.

Be familiar with the terminology of languages. Alphabet: a finite set of symbols. The alphabet is denoted by Σ (sigma). String: a finite, possibly empty sequence of symbols from an alphabet. The empty string is denoted by ε (epsilon). Language: a set, often infinite, of strings.

A language can be specified by a grammar – a generator; a system for producing all strings in the language (and no other strings) or an automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings). Many different grammars and automata may specify one language. One grammar or automaton specifies only one language.

Generally, we specify the lexical structure using regular expressions (RE), and then build the recognizer using a deterministic finite automaton (DFA). The DFA can be built by hand as we did in the Scanner class, or it can be built automatically from the RE definition as is done in Lex/flex. Generally the automatically built DFA is table driven, transitioning from state to state depending on the input stream until it reaches an accept state.

REs are defined over some alphabet Σ, and *L*(*re*) is the language generated by a particular regular expression *re* defined on Σ . In the programming world, Σ was once ASCII and now is generally Unicode to support internationalization. The basic operations defined for REs include the following, where a is an element of Σ, and r and s are regular expressions.

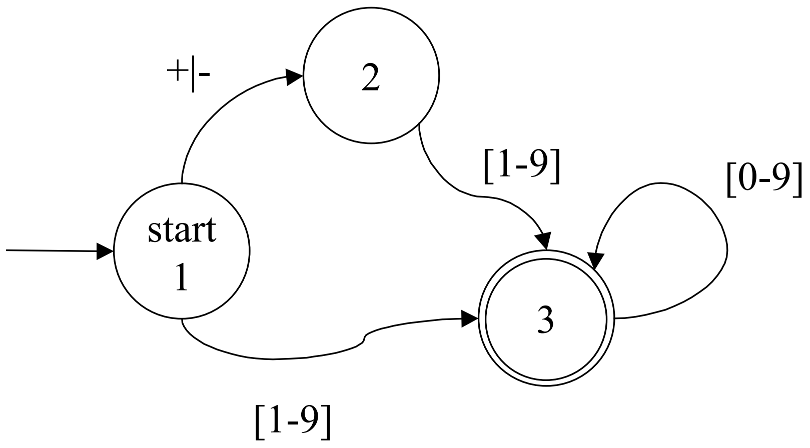
re	L(re)	Notes
ϕ	{}	empty language
ε	{ε}	empty string
a	{a}	singleton, for each a∈Σ
rs	L(r)L(s)	concatentation
r|s	L(r)∪L(s)	combination (union)
r*	L(r)*	0 or more occurences
r+	L(rr*)	1 or more occurences
r?	L(r|ε)	0 or 1 occurences
[a-z]	L(a|b|…|z)	1 character in given range
[abcxyz]	L(a|b|c|x|y|z)	1 of the given characters

Simple DFA representations can be constructed by defining a finite set of states that the DFA might be in, and identifying the transitions that will take place for a particular input when the DFA is in a given state. If the DFA is not in an accept state and end of input is reached or no transition is listed for the next input symbol, then this portion of the input is not recognized by this DFA. If the DFA is in an accept state when we reach end of input or no transition is listed for the next input symbol, then this portion of the input is accepted.

An RE that generates integers with an optional leading '+' or '-' and a leading non-zero digit followed by zero or more digits selected from the range 0 to 9 could be written as:

r = [+-]?[1-9][0-9]*

A DFA that recognizes strings in L(r) could be drawn as:



Grammar

The syntax of most modern programming languages is specified by a formal grammar, generally using the conventions of BNF (Backus-Naur Form or Backus-Normal Form).

The rules of a grammar are called *productions.* The rules contain:

Nonterminal symbols: grammar variables (*program, statement, id,* etc.)
Terminal symbols: concrete syntax that appears in programs (a, b, c, 0, 1, if, (, …)

Each production takes the form:

nonterminal ::= <sequence of terminals and nonterminals>

In a derivation, an instance of *nonterminal* can be replaced by the sequence of terminals and nonterminals on the right of the production. Often, there are two or more productions for a single nonterminal – can use either at different times.

You do not need to have memorized the D or Java grammar specifications. You should be able to read a grammar specification like the one I gave you for the D language and answer questions like: Is this input a valid *statement* according to the definitions in the given grammar? or What are all the non-terminals in this grammar?

The examples in the following table are drawn from the following grammar.

valueBlock ::= **{ get** *id***; }**
id ::= **r** | **s** | **t**

This table describes the various symbols we use in describing a grammar.

Symbol	Meaning	Usage	Example
a, b, c	elements of alphabet Σ	terminals	keyword: **get** punctuation symbols: **{ ; }** identifiers: **r s t**
w, x, y, z	elements of Σ*	strings of terminals	**{ get s; }**
A, B, C	elements of N	non-terminals	*valueBlock, id*
X, Y, Z	elements of N∪Σ	grammar symbols	*id* or **get** or **;**
α, β, γ	elements of (N∪Σ)*	strings of grammar symbols	**{ get** *id***; }**
Σ	finite set of terminals	terminals	Σ = {**get**, **r**, **s**, **t**, {, ;, } }
N	finite set of non-terminals	non-terminals	N = { *valueBlock*, *id* }
P	subset of N×(N∪Σ)*	productions	P = {<*valueBlock*,**{ get** *id***; }**>, <*id*, **r**>, <*id*,**s**>, <*id*,**t**> }

Parsing

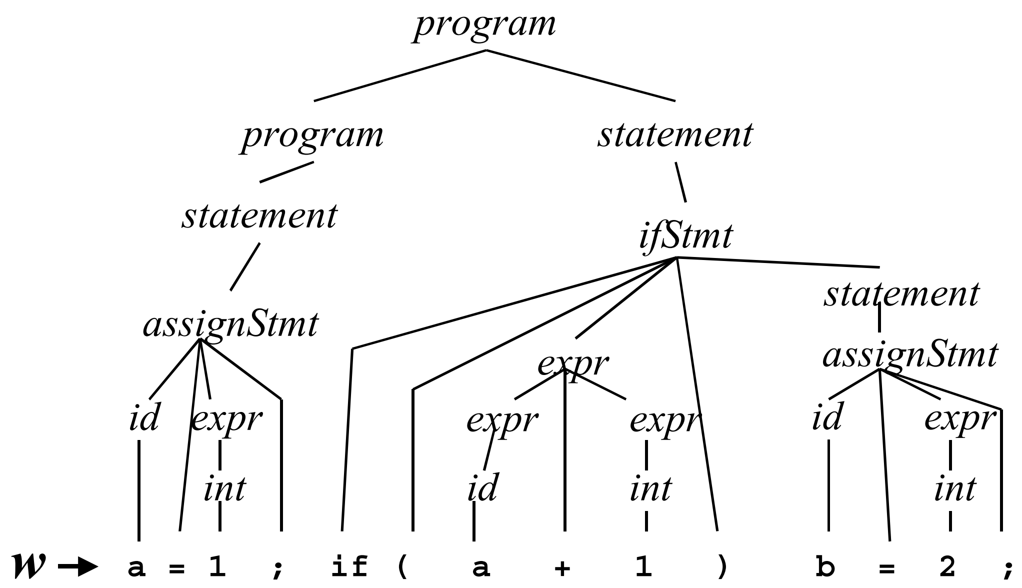
Parsing is the process of building a derivation for a given input string *w* according to the relevant grammar *G*. If the derivation can be built, then the string *w* is in the L(*G*) and we can traverse the parse tree and process *w* in a meaningful way. We may not actually build an explicit tree, but the parse proceeds as though it were doing a tree traversal.

As mentioned above in the lexical scanner section, we could do the entire job of cleaning up and tokenizing the input as part of the grammar, but that would complicate the grammar considerably. Splitting the tasks between the lexical scanner and the parser allows us to implement both functions more cleanly.

I described a simple grammar example G in lecture 19.

program ::= *statement* | *program statement*
statement ::= *assignStmt* | *ifStmt*
assignStmt ::= *id* **=** *expr* **;**
ifStmt ::= **if (** *expr* **)** *stmt*
expr ::= *id* | *int* | *expr* **+** *expr*
Id ::= **a** | **b** | **c** | **i** | **j** | **k** | **n** | **x** | **y** | **z**
int ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

If we succeed in building a derivation for a string *w* according to the grammar *G* then we can draw the parse tree for the string according to the grammar. Given a grammar *G* and a string *w,* you should be able to parse the string and draw the associated parse tree that shows each of the steps in the derivation.



Production. A production is a replacement rule. Productions can be written in a number of ways, but all of them indicate a non-terminal and the associated string of grammar symbols that it represents. Common formats for writing productions include $A\to\alpha$, $A::= \alpha$, and $<A, \alpha>$.

Derivation. Using the productions in a grammar we can move from a non-terminal A to a string of terminals w by repeatedly applying appropriate productions. For example, if $A\to\beta$ is an element of P (ie, it is a defined production), and we have the string of grammar symbols $\alpha A\gamma$, then we say "$\alpha A\gamma$ derives $\alpha\beta\gamma$", written as $\alpha A\gamma => \alpha\beta\gamma$. If there is a chain of one or more derivations that can get us from a non-terminal A to a string of terminals w, then we say that A derives w, written $A=>^*w$ and $L(A) = \{w \mid A =>^* w\}$.

Ambiguous grammar. A grammar is ambiguous is there are two or more derivations for a particular string w. Ambiguous grammars can be fixed by rewriting the grammar with more non-terminals to resolve the ambiguity or by adding "special case" checks to the parser.

Bottom up parse. LR(1). Scan the input left to right and produce a right-most derivation starting at the leaves, using 1-symbol look ahead. Such a parser is built by the parser tools Yacc/bison/CUP, and uses a shift-reduce architecture.

Top down parse. LL(1). Scan the input left to right and produce a left-most derivation starting at the root (the start symbol), using 1-symbol look ahead. Such a parser is relatively easy to build by hand, using a recursive descent architecture.

A grammar is said to be LL(1) if for all non-terminals A with productions $A\to\alpha$ and $A\to\beta$ it is true that $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$. In other words, you have to be able to tell which production to use by looking at the next symbol and knowing that it unambiguously identifies the correct production.

A recursive descent parser will enter an infinite recursion if a production is left-recursive, that is, the non-terminal on the left side of the production is the first grammar symbol on the right side of the production. One solution is to rewrite the production with an added non-terminal that can take the form of the tail of the expression or the empty string ε.

Scanner and Parser Tools

I discussed several tools in class that can be used to build scanners from well-defined regular expressions and parsers from well-defined grammars. The input to these tools is based directly on the knowledge we have developed about regular expressions and grammars, plus some arbitrary code to perform whatever functions are required by your application. Remember these tools when you need to build a scanning or parsing application in the future. (I won't ask you to use flex and bison on the final to build a parser, but you will be able to impress a future employer by doing so in an afternoon on the job.)