Compilers and Programming Languages

An Introduction

How are Languages Implemented?

- Two major strategies:
  - Interpreters (older)
  - Compilers (newer)
  - A Combination

- Interpreters run programs “as is”
  - Little or no preprocessing
- Compilers do extensive preprocessing

The Structure of a Compiler

1. Lexical Analysis
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

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

Lexical Analysis

- First step: recognize words.
  - Smallest unit above letters
  - This is a sentence.
- Lexical analyzer divides program text into "words" or "tokens"

  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 Sentences

Diagramming Sentences

- Parsing program expressions is the same
- Consider:
  - If x == y then z = 1; else z = 2;

- Diagrammed:

Semantic Analysis

- Once sentence structure is understood, we can try to understand “meaning”
  - But meaning is too hard for compilers

- Compilers perform limited analysis to catch inconsistencies.

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 in Programming

- Semantic Analysis in Programming

```
{
  int Jack = 3;
 {
   int Jack = 4;
   cout << Jack;
 }
}
```

More Semantic Analysis

- Compilers perform many semantic checks besides variable bindings

- Example:

  Jack left her homework at home.

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

Optimization

- No strong counterpart in English, but akin to editing
  - You can express the same idea using fewer words

- Automatically modify programs so that they
  - Run faster
  - Use less memory
  - In general, conserve some resource

Optimization Example

- X = Y * 0 is the same as X = 0

Code Generation

- Produces assembly code (usually)
- A translation into another language
  - Analogous to human translation

Issues

- Compiling is almost this simple, but there are many pitfalls.
- Example: How are erroneous programs handled?
- Language design has big impact on compiler
  - Determines what is easy and hard to compile
  - many trade-offs in language design

Compilers Today

- The overall structure of almost every compiler adheres to our outline
- The proportions have changed since FORTRAN
  - Early: lexing, parsing most complex, expensive
  - Today: optimization dominates all other phases, lexing and parsing are cheap

Why So Many Languages?

- Application domains have distinctive and conflicting needs.
- E.g. For scientific applications you need
  - floating point representation
  - Arrays and operations on arrays
  - Parallism
- For business Applications, you need
  - Persistence (you do not want to loose your data)
  - Report generation
  - Data analysis (i.e. you need to be able to ask questions about your data)
  - ➔ SQL

- System Languages e.g.
  - Embedded systems
  - Operating systems

- For such applications, you need
  - Very low level control over resources
  - Real time constraints, you need to be able to reason about time

Why are there new programming languages?

- Programmer trainer is the most dominant cost factor for a programming language
- ➔ Widely used languages are slow to change (because any change required educating a large number of people)
- ➔ Easy to start a new language (0 users = 0 training cost to begin with)
  - People will convert to the new language if productivity > training cost
- Languages are developed to fill a void as new technology appears e.g. mobile devices, internet, etc
- New languages look a lot like old languages (to reduce the training cost)

What is a good programming language?

- There is no universally accepted metric for language design.
- i.e.. people tend to disagree on what makes a good programming language
- 4 criteria

Language Evaluation Criteria

- **Readability**: maintenance is considered to be the major part of the software lifecycle, **and maintainability is directly related to how easily a program may be read**.

- **Writeability**: this is the measure of **how easily a language may be used to create programs**, and is closely related to readability.

- **Reliability**: a program is reliable if it **performs to its specification under all conditions.**

- **Cost:** the ultimate total cost**.**

Evaluation Criteria: Readability

- Overall simplicity
  - A manageable set of features and constructs
  - Minimal feature multiplicity (small variety)
  - Minimal operator overloading
- Orthogonality
  - A relatively **small set of primitive constructs** can be combined in a relatively small number of ways to get the desired results.
  - The more orthogonal the design, the fewer exceptions.
  - Example In IBM assembly lang there are two instructions for addition
    - A Reg1, memory_cell
    - AR Reg1, Reg2
  - However in VAX's lang there is one that is more general
    - ADDL operand1, operand2
  - VAX's instruction for addition is more orthogonal than the instructions provided by IBM; hence, it is easier for the programmer to remember (and use) than the one provided by IBM.
  - This makes it easier to learn, read and write programs in a programming language.
  - Every possible combination is legal ➔fewer exceptions

Evaluation Criteria: Readability

- Data types
  - Adequate predefined data types
- Syntax considerations
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

Evaluation Criteria: Writability

- Simplicity and orthogonality
  - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
  - The ability to define and use complex structures or operations in ways that **allow details to be ignored**
- Expressivity
  - A set of relatively convenient ways of specifying operations
  - Strength and number of operators and predefined functions

Evaluation Criteria: Reliability

- Type checking
  - Testing for type errors
- Exception handling
  - Intercept run-time errors and take corrective measures
- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
  - A language that does not support “natural” ways of expressing an algorithm will require the use of “unnatural” approaches, and hence reduced reliability

Evaluation Criteria: Cost

- Training programmers to use the language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

Language Design Trade-Offs

- Reliability vs. cost of execution
  - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs

- Readability vs. writability

  Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability

- Writability (flexibility) vs. reliability
  - Example: C++ pointers are powerful and very flexible but are unreliable

Evaluation Criteria: Others

- Portability
  - The ease with which programs can be moved from one implementation to another
- Generality
  - The applicability to a wide range of applications
- Well-definedness
  - The completeness and precision of the language's official definition

Assessment characteristics

		Criteria	
Characteristic	Readability	Writeability	Reliability
Simplicity/orthogonality	•	•	•
Control structures	•	•	•
Data types and structures	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

Influences on Language Design

- Computer Architecture
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture
- Programming Methodologies
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

Influences on Language Design

- Computer Architecture
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture
- Programming Methodologies
  - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient

The von Neumann Architecture

The von Neumann Architecture

- Fetch-execute-cycle (on a von Neumann architecture computer)

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

Programming Methodologies Influences

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  - structured programming
  - top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented
  - data abstraction
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

Language Categories

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Include languages that support object-oriented programming
  - Include scripting languages
  - Include the visual languages
  - Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- Functional
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme
- Logic
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog
- Markup/programming hybrid
  - Markup languages extended to support some programming
  - Examples: JSTL, XSLT (a language for transforming XML documents into other XML documents, or other objects such as HTML for web pages, plain text or into XSL Formatting Objects which can then be converted to PDF, PostScript and PNG.[2]

Implementation Methods

- Compilation
  - Programs are translated into machine language
- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems
  - A compromise between compilers and pure interpreters

Layered View of Computer

The operating system and language implementation are layered over machine interface of a computer

Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
  - lexical analysis: converts characters in the source program into lexical units
  - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Semantics analysis: generate intermediate code
  - code generation: machine code is generated

The Compilation Process

Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program units and linking them to a user program

Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program **instructions often can be executed much faster than the speed of the connection**; the connection speed thus results in a *bottleneck*
- Known as the *von Neumann bottleneck*; it is the **primary limiting factor** in the speed of computers

Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

Pure Interpretation Process

Hybrid Implementation Systems

- **A compromise** between compilers and pure interpreters
- A high-level language program is translated to an **intermediate language** that allows easy interpretation
- **Faster than pure interpretation**
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

Hybrid Implementation Process

Just-in-Time Implementation Systems

- Initially translate programs to an **intermediate language**
- Then **compile the intermediate language of the subprograms** into machine code when they are called
- **Machine code version is kept for subsequent calls**
- JIT systems are widely used **for Java programs**
- **.NET languages** are implemented with a JIT system

Why Study Languages and Compilers ?

1. **Increase capacity of expression**
2. Improve **understanding of program behavior**
3. Increase ability to **learn new languages**
4. Learn to **build a large and reliable** system
5. See many **basic CS concepts** a

Lexical Analysis

Part-1: Specification

Outline

- Informal sketch of lexical analysis
  - Identifies tokens in input string
- Issues in lexical analysis
  - Lookahead
  - Ambiguities
- Specifying lexers
  - Regular expressions
  - Examples of regular expressions

1. **Lexical Analysis**
2. Parsing
3. Semantic Analysis
4. Optimization
5. Code Generation

- What do we want to do? Example:

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

- The input is just a string of characters:

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

- Goal: Partition input string into substrings
  - Where the substrings are tokens

What's a Token?

- A syntactic category
  - In English:

    noun, verb, adjective, …
  - In a programming language:

    Identifier, Integer, Keyword, Whitespace, …

Tokens

- Tokens correspond to sets of strings.
- e.g.
  - Identifier: strings of letters or digits, starting with a letter
  - Integer: a non-empty string of digits
  - Keyword: “else” or “if” or “begin” or …
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs

What are Tokens For?

- Classify program substrings according to role (e.g., identifier, keyword, whitespace, …)

- Output of lexical analysis is a stream of tokens . . .
- . . . which is input to the parser
- Parser relies on token distinctions
  - An identifier is treated differently than a keyword

Example

- Input:

  X1=5

- Output

  <identifier,”x1”>, <op,”=“>, <int,”5”>

- Each pair is called a token
- Token format: <class, string>
- Or <token class, lexeme>

Designing a Lexical Analyzer: Step 1

- Define a finite set of tokens
  - Tokens describe all items of interest
  - Choice of tokens depends on language, design of parser

Example

- Recall

  \tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
- Useful tokens for this expression:

  Integer, Keyword, Relation, Identifier, Whitespace, (,), =, ;
- Note that (,), =, ;are tokens, not characters, here

Designing a Lexical Analyzer: Step 2

- Describe which strings belong to each token
- Recall:
  - Identifier: strings of letters or digits, starting with a letter
  - Integer: a non-empty string of digits
  - Keyword: “else” or “if” or “begin” or …
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs

Lexical Analyzer: Implementation

- An implementation must do two things:

1. Recognize substrings corresponding to tokens
2. Return the value or lexeme of the token
   - The lexeme is the substring

- Lexical analysis is not as easy as it sounds
- For example in FORTRAN Whitespace is insignificant
- E.g., VAR1 is the same as VA R1
- Also
  - DO 5 I = 1,25 (loop)
  - DO 5 I=1.25 (is an assignment statement)

Lexical Analysis in FORTRAN (Cont.)

- Two important points:

1. The goal is to partition the string. This is implemented by **reading left-to-right**, recognizing one token at a time
2. **"Lookahead"** may be required to decide where one token ends and the next token begins.

- FORTRAN was designed this terrible way because on punch cards machines it was easy to add whitespaces by mistake.

- Even our simple example has lookahead issues
- i vs. if
- = vs. ==

Lexical analysis in PL/I

- PL/I keywords are not reserved
- IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

- Variables
- keywords

Lexical Analysis in PL/I (Cont.)

- PL/I Declarations:
- DECLARE (ARG1,. . ., ARGN)
- Can’t tell whether DECLARE is a keyword or array reference until after the) to see if there is = for example.
- Requires arbitrary lookahead! Because we have n args. ➔ unbounded lookahead

- FORTRAN was designed in 1950’s
- PL/I was designed in 1960’s
- Things are not that bad with modern languages

- But the problems have not gone away completely.
- C++ template syntax:

  Foo<Bar>
- C++ stream syntax:

  cin >> var;
- But there is a conflict with nested templates:

  Foo<Bar<Bazz>>
- For along time C++ compilers generated a syntax error
- The only solution was to put a space between the last > >

Review

- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token of each lexeme
- Left-to-right scan => look ahead sometimes required

Next

- We still need
  - A way to describe the lexemes of each token
  - A way to resolve ambiguities
    - Is if two variables I and f?
    - Is == two equal signs =?

Regular Languages

- There are several formalisms for specifying tokens
- Regular languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

Languages

- Def. Let S be a set of characters. A language over S is a set of strings of characters drawn from S
- **Languages are sets of strings**.
- Need **some notation** for specifying which sets we want
- The standard notation for **regular languages** is regular expressions.

Regular Expressions

- Atomic Regular Expressions
  - Single character

    ‘c’={“c”}
  - Epsilon

    ε={“”}

Atomic Regular Expressions

- Union

$$A + B = \{ s \mid s \in A \text{ or } s \in B \}$$

- Concatenation

$$AB = \{ ab \mid a \in A \text{ and } b \in B \}$$

- Iteration

$$A^* = \bigcup_{i \geq 0} A^i \quad \text{where } A^i = A \ldots i \text{ times} \ldots A$$

- Def. The regular expressions over S are the smallest set of expressions including

ε	
$'c'$	where $c \in \sum$
$A + B$	where A, B are rexp over $\sum$
AB	" " "
A^*	where A is a rexp over $\sum$

Examples

- $\sum=\{0,1\}$
- $1^* = \text{“”}+1+11+111+\ldots$
- $(1+0)1 = \{ab \mid a \in 1+0 \wedge b \in 1\} = \{11,01\}$
- $0^*+1^* = \{0^i \mid i>=0\} \cup \{1^i \mid i>=0\}$
- $(0+1)^* = \cup_{i>=0} (0+1)^i = \text{“”}+ 0+1, (0+1)(0+1), \ldots, (0+1)\ldots(0+1)$
  = all strings of 0’s and 1’a

Syntax vs. Semantics

- To be careful, we should distinguish syntax (the reg. exp.) and semantics (the langs. they denote).
- Meaning function L maps syntax to semantics
- L: Exp → Sets of Strings

$$
\begin{aligned}
L(\varepsilon) &= \{\text{""}\} \\
L(\text{'}c\text{'}) &= \{\text{"}c\text{"}\} \\
L(A+B) &= L(A) \cup L(B) \\
L(AB) &= \{ab \mid a \in L(A) \text{ and } b \in L(B)\} \\
L(A^*) &= \bigcup_{i \geq 0} L(A^i)
\end{aligned}
$$

- Regular expressions are simple, almost trivial
  - But they are useful!
- Reconsider informal token descriptions . . .

Example: Keyword

- Keyword: “else”or “if”or “begin”or …

  ‘else’+ ‘if’+ ‘begin’+ . . .

  Note: ‘else’ abbreviates

  ‘e’’l’’s’’e’

Example: Integers

Integer: *a non-empty string of digits*

$$\text{digit} = \text{'0'} + \text{'1'} + \text{'2'} + \text{'3'} + \text{'4'} + \text{'5'} + \text{'6'} + \text{'7'} + \text{'8'} + \text{'9'}$$

$$\text{integer} = \text{digit digit}^*$$

Abbreviation: $A^+ = AA^*$

Example: Identifier

- Identifier: strings of letters or digits, starting with a letter

letter = 'A' + . . . + 'Z' + 'a' + . . . + 'z'

identifier = letter (letter + digit)*

Is (letter* + digit*) the same?

Example: Whitespace

- Whitespace: a non-empty sequence of blanks, newlines, and tabs

$$(\text{‘ ‘} + \text{‘\textbackslash n’} + \text{‘\textbackslash t’})^{+}$$

Example: Email Addresses

- Consider anyone@cs.stanford.edu

letter$^+$ '@' letter$^+$ '.' letter$^+$ '.' letter$^+$

or

$$\Sigma = \text{letters} \cup \{.,@\}$$

$$\text{name} = \text{letter}^+$$

$$\text{address} = \text{name '@' name '.' name '.' name}$$

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (650)-723-3232

$$
\begin{aligned}
\Sigma &= \text{digits} \cup \{-,(,)\} \\
\text{exchange} &= \text{digit}^3 \\
\text{phone} &= \text{digit}^4 \\
\text{area} &= \text{digit}^3 \\
\text{phone_number} &= \text{'('}\ \text{area}\ \text{')-'}\ \text{exchange}\ \text{'-'}\ \text{phone}
\end{aligned}
$$

Example: Unsigned Pascal Numbers

digit = '0' +'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'

digits = digit^{+}

opt_fraction = ('.' digits) + ξ ≡ (‘.’ digits) ?

opt_exponent = ('E' ('+' + '-' + ξ) digits)+ ξ

≡ (‘E’ (‘+’ + ‘-’) ? digits) ?

num = digits opt_fraction opt_exponent

Summary

- Regular expressions describe many useful languages
- Regular languages are a language specification
  - We still need an implementation
- We still need to be able to decide given a string s and a reg. exp. R, is

  $s \in L(R)$?

Lexical Analysis

Implementation: Finite Automata

Outline

- Specifying lexical structure using regular expressions
- Finite automata
  - Deterministic Finite Automata (DFAs)
  - Non-deterministic Finite Automata (NFAs)
- Implementation of regular expressions

  RegExp => NFA => DFA => Tables

Notation

- There is variation in regular expression notation

- Union: $A \mid B \equiv A + B$
- Option: $A + \varepsilon \equiv A?$
- Range: ‘a’ +’ b’ +…+’ z’ ≡ [a-z]
- Excluded range:

  complement of [a-z] ≡ [^a-z]

Regular Expressions in Lexical Specification

- Given a string s and a reg. exp. R, is

  $s \in L(R)$?

- But a yes/no answer is not enough!

- Instead: partition the input into tokens

- We adapt regular expressions to this goal

Regular Expressions => Lexical Spec.

1. Write a rexp for the lexemes of each token
   - Number = digit +
   - Keyword = ‘if’+ ‘else’+ …
   - Identifier = letter (letter + digit)*
   - OpenPar = ‘(‘
   - …

Regular Expressions => Lexical Spec.

2. Construct R, matching all lexemes for all tokens

R = Keyword + Identifier + Number + …

$= R_1 + R_2 + \ldots$

Regular Expressions => Lexical Spec.

3. Let input be $x_1 \ldots x_n$

   For $1 \leq i \leq n$ check

   $x_1 \ldots x_i \in L(R)$

4. If success, then we know that

   $x_1 \ldots x_i \in L(R_j)$ for some j

5. Remove $x_1 \ldots x_i$ from input and go to (3)

Ambiguities (1)

- There are ambiguities in the algorithm
- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$ and also
  - $x_1 \ldots x_K \in L(R)$
- e.g. = and ==
- Rule: Pick longest possible string in $L(R)$
  - The "maximal munch"
  - We as humans do that.

Ambiguities (2)

- Which token is used? What if
  - $x_1 \ldots x_i \in L(R_j)$ and also
  - $x_1 \ldots x_i \in L(R_k)$

- e.g. 'if' could be an identifier or a keyword;
- which one to choose?
- Rule: use rule listed first (j if $j < k$)
  - Treats "if" as a keyword, not an identifier
- i.e. the one listed first is given higher priority

Error Handling

- What if
  - No rule matches a prefix of input ?
- Problem: Can’t just get stuck …
- A compiler needs to give feedback to the user e.g. where the error is in the file (line number)
- Solution:
  - Write a rule matching all “bad”strings
  - Put it last (lowest priority)

Summary

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known
  - Require only single pass over the input
  - Few operations per character (table lookup)

Finite Automata

- Regular expressions = specification
- Finite automata = implementation
- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state n
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\text{state} \xrightarrow{\text{input}} \text{state}$

Finite Automata

- Transition

$$s_1 \rightarrow^a s_2$$

- Is read
- In state s_1 on input “a” go to state s_2
- If end of input and in accepting state => accept
- Otherwise => reject
  - If it terminates in state s that not a member of F
  - Or it gets stuck because there is not transition from state s1 on input a (i.e. never reaches the end of input).

Finite Automata State Graphs

- A state

- The start state

- An accepting state

- A transition

A Simple Example

- A finite automaton that accepts only “1”

Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}

And Another Example

- Alphabet {0,1}
- What language does this recognize?

Epsilon Moves

- Another kind of transition: ε-moves

- Machine can move from state A to state B without reading input

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

Execution of Finite Automata

- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take

Acceptance of NFAs

- An NFA can get into multiple states

- Input: 1 0 0
- States: {A} {A,B} {A,B,C}
- Rule: NFA accepts if it can get to a final state

NFA vs. DFA

- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are faster to execute
  - There are no choices to consider

NFA vs. DFA

- For a given language NFA can be simpler than DFA

NFA

DFA

- DFA can be exponentially larger than NFA

Regular Expressions to Finite Automata

- High-level sketch

Regular Expressions to NFA (1)

- For each kind of rexp, define an NFA
  - Notation: NFA for rexp M

- For ε

- For input a

Regular Expressions to NFA (2)

- For *AB*

- For *A* + B

Regular Expressions to NFA (3)

- For A^*

Example of RegExp -> NFA conversion

- Consider the regular expression

  (1+0)*1

- The NFA is

ε-closure of a state

- ε-closure of a state s is a set of states that consists of s and all other states that I can reach from s by making ε-moves only.
- Example
  - ε-closure(B) = {B, C, D}
  - ε-closure(G) = {G, H, I, A, B, C, D}

NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states ?
- If there are N states, the NFA must be in some subset of those N states
- How many subsets are there?
  - 2^N-1 i.e., finitely many

NFA	DFA
States : S	States : subset of S
Start state: $s \in S$	Start state: ϵ-closure(s)
Final states: F subset of S	Final state: $\{ X \mid X \cap F \neq \phi \}$
The transition function: $a(x)=\{y \mid x \in X \wedge x \xrightarrow{a} y\}$	The transition function: $X \xrightarrow{a} Y$ if $Y = \epsilon$-closure(a(X))

NFA to DFA: The Trick

- Simulate the NFA
- Each state of DFA

  = a non-empty subset of states of the NFA
- Start state

  = the set of NFA states reachable through ε-moves from NFA start state
- Add a transition $S \rightarrow^{a} S'$ to DFA iff
  - S' is the set of NFA states reachable from any state in S after seeing the input a, considering ε-moves as well

NFA -> DFA Example

Implementation

- A DFA can be implemented by a 2D table T
  - One dimension is "states"
  - Other dimension is "input symbol"
  - 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

algorithm

```
i=0;
State=0;
While(input[i]){
    State=A[state, input[i++]];
    }
```

Implementation of the NFA itself

	0	1	ξ
A			{B}
B			{ C,D }
C		{ E }	
D	{ F }		
:			

Trade off between speed and space

- DFAs
  - Faster: we are in one state at any given time.
  - less compact: there could be a large number of states 2^N-1.
- NFAs
  - slower (the loop has to deal with set of states rather than one state),
  - concise

Introduction to Parsing

Outline

- Regular languages revisited
- Parser overview
- Context-free grammars (CFG's)
- Derivations
- Ambiguity

Languages and Automata

- Formal languages are very important in CS
  - Especially in programming languages
- Regular languages
  - The weakest formal languages widely used
  - Many applications
- We will also study context-free languages, tree languages

Beyond Regular Languages

- Many languages are not regular
- Strings of balanced parentheses are not regular:

$$\{(^i)^i \mid i>=0\}$$

- There are many similar constructs in programming languages that cannot be handled with regular expressions
- E.g., nested if statements

What Can Regular Languages Express?

- So what can regular languages express?
- Consider the following FA

What can it do?

- It can tell if the number of ones in the input is divisible by 2
- i.e. it can count mod 2
- In general a FA can count mod k, where k is the number of states
- but cannot remember how many ones it has seen
- Therefore it cannot express $(^i\ )^i$
- Languages requiring counting modulo a fixed integer
- Intuition: A finite automaton that runs long enough must repeat states
- Finite automaton can’t remember # of times it has visited a particular state

The Functionality of the Parser

- Input: sequence of tokens from lexer

- Output: parse tree of the program

(But some parsers never produce a parse tree . . .)

Example

if x = y then 1 else 2 fi

- Parser input

  IF ID = ID THEN INT ELSE INT FI

- Parser output

Comparison with Lexical Analysis

Phase	*Input*	*Output*
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree

The Role of the Parser

- Not all strings of tokens are programs . . .
- . . . parser must distinguish between valid and invalid strings of tokens
- We need
  - A language for describing valid strings of tokens
  - A method for distinguishing valid from invalid strings of tokens

Context-Free Grammars

- Programming language constructs have recursive structure .

- An EXPR is

  if EXPR then EXPR else EXPR fi

  while EXPR loop EXPR pool

  …

- Context-free grammars are a natural notation for this recursive structure

CFGs (Cont.)

- A CFG consists of
  - A set of terminals T
  - A set of non-terminals N
  - A start symbol S(a non-terminal)
  - A set of productions

$$\mathbf{X \longrightarrow Y_1\ Y_2 \ldots Y_n}$$

where $X \in N$ and $Y_i \in T \cup N \cup \{\varepsilon\}$

Notational Conventions

- In these lecture notes
  - Non-terminals are written upper-case
  - Terminals are written lower-case
  - The start symbol is the left-hand side of the first production

Terminals

- Terminals are so-called because there are no rules for replacing them
- Once generated, terminals are permanent
- Terminals ought to be tokens of the language

Examples of CFGs

EXPR → if EXPR then EXPR else EXPR fi

| while EXPR loop EXPR pool

| id

Simple arithmetic expressions:

$$
\begin{aligned}
E &\rightarrow E * E \\
&\mid E + E \\
&\mid (E) \\
&\mid \text{id}
\end{aligned}
$$

The Language of a CFG

- Read productions as rules (replacement rules):

$$X \rightarrow Y_1 \ldots Y_n$$

Means X can be replaced by $Y_1 \ldots Y_n$

Key Idea

1. Begin with a string consisting of the start symbol "S"
2. Replace any non-terminal X in the string by a the right-hand side of some production

$$X \rightarrow Y_1 \ldots Y_n$$

3. Repeat (2) until there are no non-terminals in the string

The language of a CFG

- More Formally, write a single step

$$X_1 \ldots X_i \ldots X_n \rightarrow X_1 \ldots X_{i-1} Y_1 \ldots Y_m X_{i+1} \ldots X_n$$

- If there is a production

$$X_i \rightarrow Y_1 \ldots Y_m$$

The language of a CFG

- Multiple steps (0 or more steps)

$$\alpha_0 \rightarrow \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n$$

$$\alpha_0 \rightarrow^* \alpha_n \quad \text{(in 0 or more steps)}$$

The Language of a CFG

- Let G be a context-free grammar with start symbol S. Then the language of G is:

$\{ \alpha_1 \ldots \alpha_n \mid S \rightarrow^* \alpha_1 \ldots \alpha_n$ and every α_i is a terminal$\}$

- What this says is the language of a CFG is the set of strings that can be derived starting from the start symbols and contain only terminal symbols.

Examples

L(G) is the language of CFG G

Strings of balanced parentheses $\left\{ (^i)^i \mid i \geq 0 \right\}$

Two grammars:

$$S \rightarrow (S) \qquad S \rightarrow \varepsilon$$

OR

$$S \rightarrow (S) \mid \varepsilon$$

Examples of CFG

- Write a CFG that generates Even Palindrome

  $S \rightarrow aSa \mid bSb \mid \epsilon$

- Write a CFG that generates Odd Palindrome

  $S \rightarrow aSa \mid bSb \mid a \mid b$

- Write a CFG that generates Equal number of a's and b's

  $S \rightarrow aSbS \mid bSaS \mid \epsilon$

More CFG Examples

- Write a CFG that generates Equal number of a's, b's and c's

$$S \rightarrow aSbScS \mid aScSbS \mid bSaScS \mid bScSaS \mid cSaSbS \mid cSbSaS \mid \epsilon$$

Derivations and Parse Trees

- A derivation is a sequence of productions

  $S \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$

- A derivation can be drawn as a tree
  - Start symbol is the tree's root
  - For a production $X \rightarrow Y_1 \ldots Y_n$ add children $Y_1 \ldots Y_n$ to node X

Derivation Example

- Grammar

  E→ E +E | E*E| (E) | id

- String

  id * id + id

Derivation Example

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E*E+E \\
\rightarrow\ & id*E+E \\
\rightarrow\ & id*id+E \\
\rightarrow\ & id*id+id
\end{aligned}
$$

Notes on Derivations

- A parse tree has
  - Terminals at the leaves
  - Non-terminals at the interior nodes
- An in-order traversal of the leaves is the original input
- The parse tree shows the precedence of operations, the input string does not

Left-most and Right-most Derivations

- The example is a *left-most* derivation
  - At each step, replace the left-most non-terminal
- There is an equivalent notion of a *right-most* derivation

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E*E+id \\
\rightarrow\ & E*id+id \\
\rightarrow\ & id*id+id
\end{aligned}
$$

Right-most Derivation in Detail

$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E+id \\
\rightarrow\ & E*E+id \\
\rightarrow\ & E*id+id \\
\rightarrow\ & id*id+id
\end{aligned}
$$

Derivations and Parse Trees

- Note that right-most and left-most derivations have the same parse tree
- The difference is the order in which branches are added

Summary of Derivations

- We are not just interested in whether $s \in L(G)$
  - We need a parse tree for s
- A derivation defines a parse tree
  - But one parse tree may have many derivations
- Left-most and right-most derivations are important in parser implementation

Ambiguity

- Grammar

  $E \rightarrow E+E \mid E * E \mid (E) \mid id$

- String:

  id * id + id

Ambiguity

This string has two parse trees

Ambiguity

- A grammar is ambiguous if it has more than one parse tree for some string
  - Equivalently, there is more than one right-most or left-most derivation for some string

- Ambiguity is BAD
  - Leaves meaning of some programs ill-defined

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite grammar unambiguously

$$E \rightarrow E' + E \mid E'$$

$$E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E)$$

- Enforces precedence of * over +

Ambiguity: The Dangling Else

- Consider the grammar

  E → if E then E
  | if E then E else E
  | OTHER

- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  if E_1 then if E_2 then E_3 else E_4

  has two parse trees

- Typically we want the second form

The Dangling Else: A Fix

- else matches the closest unmatched then
- We can describe this in the grammar

```
E → MIF                          /* all then are matched */
  | UIF                          /* some then is unmatched */
MIF → if E then MIF else MIF
    | OTHER
UIF → if E then E
    | if E then MIF else UIF
```

- Describes the same set of strings

- The expression if E_1 then if E_2 then E_3 else E_4

- A valid parse tree (for a UIF)

- Not valid because the then expression is not a MIF

Ambiguity

- No general techniques for handling ambiguity
- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms

Precedence and Associativity Declarations

- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations
- Most tools allow precedence and associativity declarations to disambiguate grammars
- Examples …

Associativity Declarations

- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of int + int + int

- Left associativity declaration: %left +

Precedence Declarations

- Consider the grammar $E \rightarrow E + E \mid E * E \mid int$
  - And the string int + int * int

- Precedence declarations: %left +
  %left *

A General Algorithm: Recursive Descent

- Let TOKEN be the type of all special tokens: INT, OPEN, CLOSE, PLUS, TIMES.
- Let the global variable next point to the next token.
- Define boolean functions that check for a match of
  - A given token terminal

```
boolean term(Token tok)
    {return  next++==tok;}
```

- The nth production of non-terminal S;

  boolean S_n(){…}

- Try all productions of S (which succeeds if any of the productions for S matches the input)

  boolean S(){…}

Example

E→ T

E→ T+E

T→ int

T→ int*T

T→ (E)

- To start the parser
  - Initialize next to point to first token
  - Invoke E()

- Easy to implement by hand.

$E \rightarrow T \mid T + E$

$T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

(int)

```
bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }

bool E()  {TOKEN *save = next; return     (next = save, E1())
                                       || (next = save,  E2());   }
bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() { TOKEN *save = next;  return    (next = save, T1())
                                       || (next = save,  T2())
                                       || (next = save,  T3()); }
```

Problem: Left Recursion

- Given a production S➔Sα

  boolean S1(){return S()&&term(α)

  boolean S(){return S1();}
- S() goes into an infinite loop.
- Because of the left recursion
- Recursive Descent does not work in such cases
- We need to eliminate left recursion

Eliminating Left Recursion

- Consider the grammar

  $S \rightarrow S\alpha \mid \beta$

- Notice this grammar generates all strings starting with a β and followed by any number of α's
- To eliminate left recursion, we will rewrite using right recursion.
- We introduce a new non-terminal S', and write
- $S \rightarrow \beta S'$
- $S' \rightarrow \alpha S' \mid \xi$

In General

- $S \rightarrow S\alpha_1 | \ldots | S\alpha_n | ß_1 | \ldots | ß_m$
- All strings derived from S start with one of $ß_1$, …, $ß_m$ and continue with several instances of $\alpha_1 \ldots \alpha_n$.

- Rewrite as
- $S \rightarrow ß_1 S' | \ldots | ß_m S'$
- $S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \xi$

Predictive Parsing

- Like recursive descent but parser predict which production to use
  - Using look ahead (works with restricted grammar)
  - No backtracking
- Predictive parsers accept LL(K) grammars
  - Left to right
  - Left most derivation
  - K tokens look ahead (usually k=1)

LL(1)

- In LL(1)
  - At each step only one choice of production
  - Given wAb on input t, there is at most one production that can be used

Refactoring

- Consider the grammar
- E➔T+E|T
- T➔int|int*T|(E)
- It is hard to predict which production to use
  - There are two production that can be used for E
  - and two productions that can be used for T (the two that begin with int)
  - This grammar is not acceptable for predictive LL(1) parsing

- We need to left-factor the grammar
- By eliminating common prefixes
- Example

  E➔T+E|T
- Becomes

  E➔TX

  X➔+E|ξ

- T➔int|int*T|(E)
- Becomes
  - T➔ int Y|(E)
  - Y➔*T|ξ

- What we did
  - We factored out the common prefix (which is T in the first example and int in the second)
  - We introduced a new nonterminal (X in the first example and Y in the second)
  - We used one production for T and
  - one for the new non-terminal that list all choices

Predictive Parsing

- Like recursive descent but parser predict which production to use
  - Using look ahead (works with restricted grammar)
  - No backtracking
- Predictive parsers accept LL(K) grammars
  - Left to right
  - Left most derivation
  - K tokens look ahead (usually k=1)

LL(1)

- In LL(1)
  - At each step only one choice of production
  - Given wAb on input t, there is at most one production that can be used

Refactoring

- Consider the grammar
- E➔T+E|T
- T➔int|int*T|(E)
- It is hard to predict which production to use
  - There are two production that can be used for E
  - and two productions that can be used for T (the two that begin with int)
  - This grammar is not acceptable for predictive LL(1) parsing

- We need to left-factor the grammar
- By eliminating common prefixes
- Example

  E➔T+E|T
- Becomes

  E➔TX

  X➔+E|ξ

- T➔int|int*T|(E)
- Becomes
  - T➔ int Y|(E)
  - Y➔*T|ξ

- What we did
  - We factored out the common prefix (which is T in the first example and int in the second)
  - We introduced a new nonterminal (X in the first example and Y in the second)
  - We used one production for T and
  - one for the new non-terminal that list all choices

- Left factored grammar

$E \rightarrow TX$ $\quad$ $X \rightarrow +E \mid \xi$

$T \rightarrow (E) \mid Y$ $\quad$ $Y \rightarrow *T \mid \xi$

- The LL(1) parsing table

	int	*	+	(	)	$
E	T X			T X		
X			+ E		ε	ε
T	int Y			(E)		
Y		* T	ε		ε	ε

- The leftmost column represents the leftmost. non-terminal symbol in a derivation
- The top row represents the next input token.
- For example the [E, int] entry, says
  - When current non-terminal is E and next input is int, use production $E \rightarrow TX$

- Notice blank entries represent errors
- For example entry [E, *] is blank
- Indicating that there is no production to use for E to get successful parsing, in the input token is *.

LL(1) algorithm

- A method similar to recursive descent except
  - For the leftmost non-terminal S
  - We look at the next input token a
  - And choose the production shown at [S,a]
- Use a stack to record leaf nodes (frontiers) of the parse tree
- The top of stack is the leftmost pending terminal or non-terminal
- Reject on reaching error state
- Accept on end of input and empty stack

The LL(1) Algorithm

- Suppose a grammar has start symbol S and LL(1) parsing table T. We want to parse string ω
- Initialize a stack containing S$.
- Repeat until the stack is empty:
  - Let the next character of ω be t
  - If the top of the stack is a terminal r:
    - If r and t don't match, report an error.
    - Otherwise consume the character t and pop r from the stack.
  - Otherwise, the top of the stack is a nonterminal A:
    - If T[A, t] is undefined, report an error.
    - Replace the top of the stack with T[A, t].

Example

- Let's parse int*int, drawing the parse tree at each step.

LL(1) Parsing Example

Stack	Input	Action
E $	int * int $	T X
T X $	int * int $	int Y
int Y X $	int * int $	terminal
Y X $	* int $	* T
* T X $	* int $	terminal
T X $	int $	int Y
int Y X $	int $	terminal
Y X $	$	ε
X $	$	ε
$	$	ACCEPT

Constructing the Parse Table

- Consider
  - A non-terminal A
  - Production $A \rightarrow \alpha$
  - And an input token t
  - We want to know the conditions under which we can make the move $T[A,t]=\alpha$
- We make the move $T[A,t]=\alpha$ in two situations

1. If $\alpha \rightarrow^{*} t\beta$ i.e. α can derive a t in the first position

In this case we say that $t \in First(\alpha)$

And the move $T[A,t]=\alpha$ is reasonable

2. Or if $A \rightarrow \alpha$, and

   $\alpha \rightarrow^{*} \xi$ (i.e. α can disappear), and

   $S \rightarrow^{*} \beta\ A\ t\ \delta$ (notice since α can disappear so does A)

- Notice that this is useful if t can follow A and A can disappear.
- In other words A does not derive t but t follows A.

- This case we say $t \in Follow(A)$

First Sets

- Def.
- First(X)={t | X→* tα} ˅ { ξ | X→* ξ }
- Notice that the last part is there because we need to keep track of whether or not X can produce ξ.
- Algorithm :

1. If t is a terminal

   First (t) = { t }

2. If X is non-terminal, then $\xi \in$ First(X)
   1. If $X \rightarrow \xi$
   2. Or if $X \rightarrow A_1,\ldots A_n$ and $\xi \in$ First(A_i) for $1 \leq i \leq n$
      i.e. if $A_1,\ldots A_n$ can disappear by producing ξ
3. First (α) is a subset of First(X) if

   $X \rightarrow A_1,\ldots A_n\ \alpha$

   and $\xi \in$ First(A_i) for $1 \leq i \leq n$

   (i.e. $A_1,\ldots A_n$ can all disappear)

Example on First Sets

- E → T X
- T → (E) | int Y

X → +E | ξ

Y → * T| ξ

1. Terminals

First(+)={+}

First(*)={*}

First(()={(}

First())={)}

First(int)={int}

2. Non-terminals
   - First(E)
     1. Since E → TX , then First(E) is a super set of First(T) and First(T) = { (, int }
     2. Notice if T →* ξ then First(E) is a super set of First(X) but this is not the case since First(T) does not contain ξ

     Therefore, First(E) = First(T)= { (, int }
   - First(X)= { + , ξ }
   - First(Y)= { * , ξ }

Follow Sets

- Notice Follow(X) is not about what X produces but rather about where X appears.
- Definition

  $$\text{Follow}(X)=\{ t \mid S \rightarrow^{*} \beta X t \delta \}$$

- Intuition
  - If $X \rightarrow A\beta$ then
    - First(B) is a subset of Follow(A)
    - Follow(X) is a subset of Follow(B) (i.e., anything that can come after X is included in the follow of B)

- If $X \rightarrow A\beta$ and $\beta \rightarrow^{*} \xi$

  then Follow(X) is a subset of Follow(A)

  (i.e., anything that can come after X is included in Follow(A))

- If S is the start symbol, then $ ϵ Follow(S)

  (we always add $ in the Follow of the start symbol)

  Because it is what we have when we runout of input)

Algorithm

1. $ ϵ Follow(S), where S is the start symbol
2. For each production A → α X β

   First(β) – { ξ } is a subset of Follow(X)

   (notice that we exclude ξ , because ξ is never in a follow set)

3. For each production A → α X β

   if ξ ϵ First(β) (i.e., β can completely disappear)

   then whatever is in Follow(A) is also in Follow(X)

   i.e., Follow(A) is a subset of Follow(X)

Example

- $E \rightarrow T\ X$ $\qquad$ $X \rightarrow +E \mid \xi$
- $T \rightarrow (E) \mid int\ Y$ $\qquad$ $Y \rightarrow *\ T \mid \xi$
- Remember to determine the follow of X we need to look at where X appears
- Follow(E)
  1. Since E is a start symbol, $ is ϵ Follow(E)
  2. Since T → (E) , then) is ϵ Follow(E)
  3. Since X → +E, then anything that is in the follow of X is also in the follow of E (i.e. Follow(X) is a subset of Follow(E))
  4. Since E → T X then any thing that is in the follow of E is also in the follow of X (i.e. Follow(E) is a subset of Follow(X))
  5. From 3 and 4 we conclude that Follow(E)=Follow(X)
  6. Both are { $,) }

- Follow(T)
  1. Since E → T X, then Follow(T) includes First(X) (which is {+, ξ} but we must exclude ξ).
  2. Since X → ξ , Follow(T) must include follow(E)
  3. (i.e. Follow(E) is a subset of Follow(T))
  4. Since T also appears in Y → * T then Follow(T) includes Follow(Y) (Follow(Y) is a subset of Follow(T)
  5. But notice that T → int Y so Follow(T) is also a subset of Follow(Y)
  6. From 4 and 5, we conclude that Follow(T)=Follow(Y)={ +, $,) }

Follow of Terminal Symbols

- Follow(‘(‘)
  - Since ‘(‘ appears in T → (E), then Follow(‘(‘) includes First(E) (i.e. it includes { (, int })
  - Since ‘(‘ does not appear anywhere else
  - Follow(‘(‘)= { (, int }

- Follow(‘)’)
  - Since ‘)’ appears only in T → (E),

    Follow(‘)’) must include only Follow(T)
  - Follow(‘)’) = {+, $,)}

- Follow(‘+’)
  - Since + is only used in X → +E

    Follow(‘+’) includes First(E), which is { (, int} .
  - Notice the E cannot produce ξ
  - Follow(‘+’) = { (, int}

- Follow(‘*’)
  - Since ‘*’ is only used in Y → * T

    Follow(‘*’) includes First(T), which is { (, int}
  - Since T cannot got to ξ then that is it
  - Follow(‘*’)= { (, int}

- Follow(int)
  - Since int only appears in T → int Y
  - Follow(int) includes First(Y) which is {*}
  - But since Y→ξ, Y can completely diappear therefore, Follow(int) must include Follow(T) (which is {+,$,)})
  - Follow(int)={*, +,$,)}

Putting Together First sets and Follow Sets to Construct an LL(1) table

- For each production A→ α in G do
  - For each terminal t ϵ First(α) do
    - T[A,t]= α because obviously would is useful here
  - If ξ ϵ First(α), for each t ϵ Follow(A) do
    - T[A,t]= α because α can completely disappear and consequently A disappears.
  - If ξ ϵ First(α) and $ ϵ Follow(A) do
    - T[A,$]= α This is useful when we ran out of input because the only hope would be is to get rid of whatever is on the stack.

Example

- E → T X
- T → (E) | int Y

X → +E | ξ

Y → * T| ξ

	(	)	+	*	int	$
E	TX				TX	
T	(E)				int Y	
X		ξ	+E			ξ
Y		ξ	ξ	*T		ξ

T[E,( ] = T[E, int] = TX because (and int are in the First of TX

T[T, ( ] = (E) because (is in the First((E))

T[T, int] = int Y because int is in the First(int Y)

T[X, +] = +E because + is in the First(+E)

T[Y, *] = *T because * is in the First(*T)

T[X,)] = ξ because X → ξ and) is in the Follow(X)

T[X,$]= ξ because X → ξ and $ is in the Follow(X)

T[Y,)] = ξ because Y → ξ and) is in the follow(Y)

T[Y, +] = ξ because Y → ξ and + is in the follow(Y)

T[Y, $] = ξ because Y → ξ and $ is in the follow(Y)

Not all grammars are LL(1) grammars

- Example:
- S→ Sa| b
- First(S) = {b}
- Follow(S) = { $, a}
- Let's try to construct an LL(1) table

	a	**b**	**$**
S		b Sa	

- Notice that we have multiply defined entry
- i.e., 2 possible moves to make, not deterministic
- We conclude that the grammar is not LL(1) grammar

- If an entry is multiply defined, the G is not an LL(1) grammar
- The list includes (but not limited to)
  - Any grammar that is not left factored
  - Any grammar that contains left recursion (the above example)
  - Any grammar that is ambiguous
  - Any grammar that requires more than 1 look ahead token
- Remember the above list is not comprehensive
- The only way to make sure is by trying to construct an LL(1) parsing table

- Most programming languages CFGs are not LL(1).
- LL(1) grammars are to weak to capture many interesting constructs in PLs
- The solution will build up on what we have learned so far.

Bottom Up Parsing

- More general than deterministic top-down parsing
  - Just as efficient
  - Uses the same ideas
- The method used by most compiler generation tools
- +ve: do not need left factored grammar
- For example the following grammar is OK

  E→T+E|T

  T → int*T | int |(E)
- A grammar is OK provided that it is unambiguous

What is Bottom-Up Parsing?

- Idea: Apply productions in reverse to convert the user's program to the start symbol.
- A left-to-right, bottom-up parse is a rightmost derivation traced in reverse (as we will see).

One View of a Bottom-Up Parse

E → T
E → E + T
T → int
T → (E)

A Second View of a Bottom-Up Parse

$E \rightarrow T$
$E \rightarrow E + T$
$T \rightarrow \texttt{int}$
$T \rightarrow (E)$

$$
\begin{aligned}
& \texttt{int} + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & T + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (\texttt{int} + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (T + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (E + \texttt{int} + \texttt{int}) \\
\Rightarrow\ & E + (E + T + \texttt{int}) \\
\Rightarrow\ & E + (E + \texttt{int}) \\
\Rightarrow\ & E + (E + T) \\
\Rightarrow\ & E + (E) \\
\Rightarrow\ & E + T \\
\Rightarrow\ & E
\end{aligned}
$$

A Third View of a Bottom-Up Parse

```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```

Each step in this bottom-up parse is called a **reduction**. We **reduce** a substring of the sentential form back to a nonterminal.

A Third View of a Bottom-Up Parse

```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```

A Third View of a Bottom-Up Parse

```
  int + (int + int + int)
⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E
```

A Third View of a Bottom-Up Parse

⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

A Third View of a Bottom-Up Parse

⇒ T + (int + int + int)
⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

A Third View of a Bottom-Up Parse

⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

11.03 X 8.26 inch

A Third View of a Bottom-Up Parse

⇒ E + (int + int + int)
⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

11.03 X 8.26 inch

A Third View of a Bottom-Up Parse

⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

A Third View of a Bottom-Up Parse

⇒ E + (T + int + int)
⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

11.03 X 8.26 inch

A Third View of a Bottom-Up Parse

⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

11.03 X 8.26 inch

A Third View of a Bottom-Up Parse

⇒ E + (E + int + int)
⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

A Third View of a Bottom-Up Parse

⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

⇒ E + (E + T + int)
⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

A Third View of a Bottom-Up Parse

⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

⇒ E + (E + int)
⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

⇒ **E + (E + T)**
⇒ **E + (E)**
⇒ **E + T**
⇒ **E**

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

⇒ E + (E + T)
⇒ E + (E)
⇒ E + T
⇒ E

int	+	(	int	+	int	+	int	)

11.03 X 8.26 inch

A Third View of a Bottom-Up Parse

⇒ **E + (E)**
⇒ **E + T**
⇒ **E**

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

⇒ **E + (E)**
⇒ **E + T**
⇒ **E**

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

⇒ **E + T**
⇒ **E**

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

⇒ **E + T**

⇒ **E**

int	+	(	int	+	int	+	int	)

A Third View of a Bottom-Up Parse

E

$\Rightarrow$ **E**

int	+	(	int	+	int	+	int	)

Observation

- Read the productions in reverse (from bottom to top)
- This is a rightmost derivation!

int * int + int	$T \rightarrow$ int
int * T + int	$T \rightarrow$ int * T
T + int	$T \rightarrow$ int
T + T	$E \rightarrow T$
T + E	$E \rightarrow T + E$
E	

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse

A Bottom-up Parse in Detail (1)

int * int + int

int * int + int

A Bottom-up Parse in Detail (2)

int * int + int

int * T + int

A Bottom-up Parse in Detail (3)

int * int + int

int * T + int

T + int

A Bottom-up Parse in Detail (4)

int * int + int

int * T + int

T + int

T + T

A Bottom-up Parse in Detail (5)

int * int + int

int * T + int

T + int

T + T

T + E

A Bottom-up Parse in Detail (6)

int * int + int

int * T + int

T + int

T + T

T + E

E

Shift Reduce Parsing

- The main strategy used by bottom up parsers
- Recall that a bottom-up parser traces a rightmost derivation in reverse
- An important consequence
  - Let αβω be a step of a bottom-up parse
  - Assume the next reduction is by X→ β
  - Then ω is a string of terminals, otherwise the reduction we just did was not for the rightmost terminal.
- The general idea: split string into two substrings
  - Right substring is yet unexamined by parsing
  - The Left substring has terminals and non-terminals
  - The left substring is our work area (where we should search for handles)
  - The dividing point is marked by a |

Two Main kinds of actions

1. Shift: Move | one place to the right
   - i.e., shifts a terminal to the left string
   - e.g. ABC|xyz➔ ABCx|yz

2. Reduce: apply an inverse production at the right end of the left string
   - If A→ xy is a production, then
     Cbxy | ijk ➔ CbA| ijk

- When to shift and when to reduce is another story

The Example with Shift-Reduce Parsing

|int * int + int	shift
int | * int + int	shift
int * | int + int	shift
int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T | + int	shift
T + | int	shift
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	

Bottom-Up Parsing Using a Stack

- Left string can be implemented by a stack
  - Shift
    - pushes a terminal on the stack
  - Reduce
    - Pop symbols off of the stack (the right hand side of a production)
    - Pushes a non-terminal on the stack (the left hand side of a production)

Conflicts

- In a given state, more than one action (shift or reduce) may lead to a parse tree
- Two main kinds of conflicts:
  - If it is legal to shift or reduce, there is a shift-reduce conflict
    - Not very good, but it is easy to rewrite the grammar to remove it.
  - If it is legal to reduce by two different productions, there is a reduce-reduce conflict.
    - This is bad because it indicates that something is wrong with the grammar

Handles

- How do we decide when to shift or reduce?
- The leftmost reduction is not always the best thing to do
- 2 Examples

A Detail about Handles

E → F
E → E + F
F → F * T
F → T
T → int
T → (E)

Another Example

- Example grammar:
  - E→ T+E | T
  - T → int * T | int | (E)
  - Consider step int | * int + int
    - We could reduce by T → int giving T| *int+int
    - A fatal mistake
      - Since no production can handle T*
      - There would be no way to reduce to the start symbol E

Handles

- A handle is a reduction that **allows further reductions back to the start symbol**
- $S \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\omega$, then $\alpha\beta$ is the handle simply because it is **not a mistake it allowed us to go back to the start symbol**.
- The handle of a parse tree T is **the leftmost complete cluster of leaf nodes.**

- A left-to-right, bottom-up parse works by
  - iteratively searching for a handle, then
  - reducing the handle.

Finding Handles

- **Where** do we look for handles?
  - Where in the string might the handle be?
- **How do we search** for possible handles?
  - Once we know where to search, how do we identify candidate handles?
  - What algorithm do we use to try to discover a handle?
- **How do we recognize handles**?
  - Once we've found a candidate handle, how do we confirm that it is correct (i.e., the handle?)

Recognizing Handles

- There are **no known efficient algorithms** to recognize handles

- But, there are **good heuristics for guessing handles**
- **On some CFGs, the heuristics always guess correctly**

- It is **not obvious how to detect** handles
- At each step the parser **sees only the stack**, not the entire input;
- It sees α where **α is a viable prefix** if there is an ω such that **α| ω is a state of a shift-reduce** parser.
- Recall that α is on the stack while ω is the unseen input.

Viable Prefix

- A viable prefix because is a **prefix of the handle**
- In other words: it **does not extend past the right end** of the handle
- As long as a parser has viable prefixes on the stack **no parsing error has been detected**.

Important Fact

- For any grammar, **the set of viable prefixes is a regular language**.
- Therefore they **can be recognized by a finite automata**
- The basis for many **compiler generation tools**.
- We will see **how to construct such a FA**.
- But first we need **a few more definitions**

An item

- An item is **a production with a "." somewhere on the rhs.**
- The items for T → (E) are

  T → .(E)

  T → (.E)

  T → (E.)

  T → (E).
- The only item for X → ξ is X → .
- **Items are often called "LR(0) items"**

The problem in recognizing viable prefixes

- is that the stack has only **bits and pieces** of the rhs of productions
  - If it had a complete rhs, we could reduce
- In any successful parse **theses bits and pieces are always prefixes of rhs of productions**.

- Consider the input (int)

  E → T + E | T

  T → int*T | int | (E)

  – Then (E|) is a **valid state of a shift-reduce** parse
  – (E is a prefix of the rhs of T → (E)
    - Notice that it will be reduced after the next shift
  – Item T → (E.) says that so far we have seen (E of this production and hope to see)
    - i.e. **no parsing errors so far**

The structure of the stack

- The stack does not contain an arbitrary string of symbols.
- The stack may have many prefixes or rhs's
- $Prefix_1\ Prefix_2 \ldots Prefix_{n-1}\ Prefix_n$

- Let $Prefix_i$ be a prefix of rhs of $X_i \rightarrow \alpha_i$
  - $Prefix_i$ will eventually reduce to X_i
  - The missing part of α_{i-1} starts with X_i
  - i.e., there is an $X_{i-1} \rightarrow Prefix_{i-1}\ X_i\ \beta$ for some β
  - Recursively, $Prefix_{k+1} \ldots Prefix_n$ eventually reduces to the missing part of α_k

An Example

- Consider the grammar

  E → T + E | T

  T → int*T | int | (E)

- And the string (int * int)
- (int * | int) is a state of a shift-reduce parse
- The stack contents from bottom-to-top is

  "(" which is a prefix of the rhs of T → (E)

  "ξ" which is prefix of the rhs of E → T

  "int *" which is a prefix of the rhs of T → int * T

- The stack of items

  T → (.E)

  E → .T

  T → int * .T

- Which says
  - We have seen "(" of T → (E)
  - We have seen ξ of E → T
  - We have seen int* of T → int * T

- **To recognize viable prefixes, we must**
  - **Recognize a sequence** of partial rhs's of productions, where
  - Each partial rhs can eventually **reduce to part of the missing suffix of its predecessor**

An Algorithm for recognizing Viable prefixes

- Recall that the set of **viable prefixes are regular**, so what we are going to do is to **construct a NFA that recognizes** them.
- The input of the NFA is the stack.
  - **It will be read bottom-up**
- The **output is**
  - **yes if it is a viable prefix and**
  - **no if it is not**.
- **The states of the NFA are the items of the grammar.**

Algorithm:

1. Add a dummy production S’ → S to G
   - this makes S’ the new start symbol and
   - makes sure that there is one production for the new start symbol
2. The NFA **states are the items of G**
   - Including the extra production
3. For item E → α.Xβ add transition
   - (i.e. so far we have seen α on the stack)
   - So if x is the next symbol on the stack (above α) then we can make this transition
   - E → α.Xβ $\Rightarrow^{x}$ E → αX.β where X a terminal or non terminal (i.e., a move the NFA can make)
4. For item E → α.Xβ and for every production X →ϒ
   - where X is a non-terminal
   - and what is on the stack can eventually be reduced to x
   - So we can make the transition
   - E → α.Xβ $\Rightarrow^{\xi}$ X → .ϒ
5. Every state is an accepting state
6. Start State is S’ → .S

Example

S’ → E (the extra production)

E → T + E | T

T → int*T | int | (E)

The start state is the extra production

$S' \rightarrow . E$

What transitions can we make depends on what can be on the stack:

There can be an E on the stack or something derived from E so we need three transitions

Notice the dot on the left of T, indicating that we are hoping to see T on the stack next.

Notice the dot on the left of T, indicating that we are hoping to see T on the stack next.

The transitions for E→.T

If we do not see a T, we may see something that can be derived from T

Notice the dot on the right indicates that we are ready to perform a reduction.

The transitions for E → .T+E

The transitions for T → .(E):

One possible transition that is if we see "(" on the stack.

The transition for T → (.E)

The transition of T → (E.)

We can make a reduction

The transition for E → T.+E

Transitions for E → T+.E

Transitions for T → .int

Transitions for T → .int * T

Transitions for T → int. * T

Transitions for T→ int *.T

An Equivalent DFA

- Notice: that each item is state and
- the NFA can be in any of these states

- The states of the DFA are “canonical collections of items”
- Or “canonical collections of LR(0) items”
- Item $X \rightarrow \beta.\Upsilon$ is valid for a viable prefix $\alpha\beta$ if

$$S' \rightarrow^* \alpha X \omega \rightarrow \alpha\beta\Upsilon\omega$$

by a rightmost derivation

- After parsing $\alpha\beta$, the valid items are the possible tops of the stack of items.

Valid Items

- An item is often valid for many prefixes
- Example: The item T → (.E) is valid for prefixes

  (

  ((

  (((

  ((((

  (((((

  …

SLR Parsing Algorithm: Simple LR parsing

- LR(0) parsing: Assume
  - Stack contains α
  - Next input is t
  - DFA on input α terminates in state s
- Reduce by X → β if
  - S contains item X → β. (i.e. we have seen a complete rhs)
- Shift if
  - S contains item X → β.tω
  - i.e. s has a transition labeled t

2 kinds of problems

- LR(0) may not be able to decide what to do in two situations
- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
  - $X \rightarrow \beta.$ and $Y \rightarrow \omega.$ (two possible reduce actions)
- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
  - $X \rightarrow \beta.$ and $Y \rightarrow \omega.t\delta$
    - (i.e. a reduce is possible and a shift is also possible)

A shift reduce conflict
A shift reduce conflict
S' → E .
S' → . E
E → . T
E → .T + E
T → .(E)
T → .int * T
T → .int
E → T.
E → T. + E
T → int. * T
T → int.
T → int * .T
T → .(E)
T → .int * T
T → .int
T → int * T.
T → (E.)
T → (E).
E → T + . E
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int
E → T + E.
T → (. E)
E → .T
E → .T + E
T → .(E)
T → .int * T
T → .int
E
T
int
*
(
)
+

SLR Parsing

- SLR = “Simple LR”
- SLR improves on LR(0) by adding shift/reduce heuristics that
- will help us determine when to reduce and when to shift
  - Fewer states have conflicts

SLR Parsing

- Assumptions
  - Stack contains α
  - Next input is t
  - DFA on input α terminates in state s.
- Reduce by X → β if
  - S contains X → β.
  - And
  - t ϵ Follow(X) (where t is the next input)
- Shift if
  - s contains item X → β.tω

- If we still have conflicts after applying these rules, then the grammar is not SLR.
- The rules are heuristics for detecting handles
  - The SLR grammars are those where the heuristics detect exactly the handles.

Parsing Example

We will reduce if the next input is in the Follow(E) which is {$,)}
and shift if the next input is +

We will reduce if the next input is in the Follow(T) which is {$,),+}
and shift if the next input is *

- Notice that all conflicts are resolved in the above grammar so it is an SLR grammar.
- But many grammars are not SLR
  - These include all ambiguous grammar.
- We can parse more grammars by using precedence declarations
  - Instructions for resolving conflicts

- Consider the grammar
- E→ E+E | E*E | (E) | int
- The DFA for this grammar contains a state with the following items:
  - E→ E*E. And E→ E.+E
  - Shift/reduce conflict.
- Declaring that * has higher precedence than + resolves this conflict in favor of reducing.
- So we will not do the shift.

SLR Parsing algorithm

- Let M be a DFA for viable prefixes of G
- Let $|x_1 \ldots X_n\$$ be initial configuration (the stack is empty)
- Repeat until configuration is $S|\$$ (i.e. until all input has been consumed)
  - Let $\alpha|\omega$ be current configuration
  - Run M on current stack α
  - If M rejects α, report parsing error
    - Stack α is not a viable prefix
  - If M accepts α and ends in a state with items I, let a be next input
    - Shift if $X \rightarrow \beta.a\Upsilon \in I$
    - Reduce if $X \rightarrow \beta. \in I$ and $a \in Follow(X)$
    - Report parsing error if neither applies

A Parsing Example: int * int$

Configuration	DFA Halt State	Action
|int*int$	1	Shift
int| * int$	3 (because * is not in Follow(T)	Shift
int * | int $	11	shift
int * int |$	3 Because $ϵ Follow(T)	Reduce T → int
int * T |$	4 Because $ϵ Follow(T)	Reduce T → int*T
T|$	5 Because $ϵ Follow(E)	Reudce E → T
E|$		accept (since E is the start symbol)

Semantic Analysis

5.0 Semantic Analysis

Where We Are?

- Program is lexically well-formed:
  - Identifiers have valid names.
  - Strings are properly terminated.
  - No stray characters.
- Program is syntactically well-formed:
  - Class declarations have the correct structure.
  - Expressions are syntactically valid.
- Does this mean that the program is legal (valid)?

5.0 Semantic Analysis

Consider the following program:
It is syntactically correct, but is it error free?

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

Interface not declared

Wrong type

Can't multiply strings

Variable not declared

Can't redefine functions

Can't add void

No main function

- Semantic analysis is our last line of defense.
- Parsing cannot catch all errors
- This is because CFG are **not expressive enough** to describe everything we are interested in in a language.
- **i.e., some language constructs are not context free**.

5.0 Semantic Analysis

Limitations of CFGs

- Using CFGs:
- How would you prevent **duplicate class definitions**?
- How would you differentiate **variables of one type from variables of another** type?
- How would you **ensure classes implement all interface methods**?
- For most programming languages, these are **provably impossible**.

Implementing Semantic Analysis

- Attribute Grammars
  - Augment rules to do checking during parsing.
  - Approach suggested in the Compilers book.
  - Has its limitations; more on that later.
- Recursive AST Walk
  - Construct the AST, then use virtual functions and recursion to explore the tree.
  - The approach we'll take in this class.

Abstract Syntax Tree: AST

- Much of the semantic analysis can be expressed as a recursive descent of an AST.
- When we traverse an AST some operations are performed on a node before we process its children and some operations are performed after we process its children
  - Before: process an AST node n
  - Recurse: Process the children of n
  - After: Finish processing the AST node n
- This is called **Recursive Decent Traversal of a Tree**
  - Sometimes we process a node before its children, sometimes after, and sometimes both.
- When performing semantic analysis on a portion of the AST, we need to know **which identifiers are defined**

Types of Checks

- Scope-Checking
  - How can we tell **what object a particular identifier refers to**?
  - How do we **store this information**?

- Type-Checking
  - How can we tell whether **expressions have valid types**?
  - How do we know all **function calls have valid arguments**?

Scope

- The same name in a program **may refer to fundamentally different things**:
- This is perfectly legal Java code:

```
public class A {
    char A;
    A A(A A) {
        A.A = 'A';
        return A((A) A);
    }
}
```

```
public class A{
    char A;
    A A(A A) {
        A.A= 'A';
        return A((A) A);
    }
}
```

- This is perfectly legal C++ code:

```
int Awful() {
    int x= 137;
    {
        string x= "Scope!"
        if (float x= 0)
            double x= x;
    }
    if (x== 137) cout << "Y";
}
```

Scope

- The scope of an entity is the **set of locations** in a program where that **entity's name refers to that entity.**

- The introduction of new variables into scope **may hide older variables**.

- **How do we keep track of what's visible**?

Symbol Tables

- A symbol table is a **mapping from a name to the thing that name refers to**.
- As we run our **semantic analysis, continuously update the symbol table** with information about what is in scope.
- Questions:
  - What does this look like in practice?
  - **What operations** need to be defined on it?
  - How do we **implement** it?

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5
y	9

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5
y	9

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5
y	9

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5
y	9

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

Symbol Table	
x	0
z	1
x	2
y	2
x	5
z	5

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x@5, y@2, z@5);
16:   }
17: }
```

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x@5, y@2, z@5);
16:   }
17: }
```

Symbol Tables: The Intuition

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x@2, y@2, z@1);
 4:   {
 5:     int x, z;
 6:     z@5 = y@2;
 7:     x@5 = z@5;
 8:     {
 9:       int y = x@5;
10:       {
11:         printf("%d,%d,%d\n", x@5, y@9, z@5);
12:       }
13:       printf("%d,%d,%d\n", x@5, y@9, z@5);
14:     }
15:     printf("%d,%d,%d\n", x@5, y@2, z@5);
16:   }
17: }
```

Symbol Table	
x	0
z	1

Symbol Table Operations

- Typically implemented as a **stack of maps**.
- Each map corresponds to a particular scope.
- Stack allows for easy “enter” and “exit” operations.
- Symbol table operations are
  - **Push scope**: Enter a new scope.
  - **Pop scope**: Leave a scope, discarding all declarations in it.
  - **Insert symbol**: Add a new entry to the current scope.
  - **Lookup symbol**: Find what a name corresponds to.

Using a Symbol Table

- To process a portion of the program that creates a scope (block statements, function calls, classes, etc.)
  - Enter a new scope.
  - Add all variable declarations to the symbol table.
  - Process the body of the block/function/class.
  - Exit the scope.
- Much of semantic analysis is defined in terms of recursive AST traversals like this.

Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Another View of Symbol Tables

```
 0: int x;
 1: int y;
 2: int MyFunction(int x, int y)
 3: {
 4:   int w, z;
 5:   {
 6:     int y;
 7:   }
 8:   {
 9:     int w;
10:   }
11: }
```

Spaghetti Stacks

- Treat the symbol table as a linked structure of scopes.
- Each scope stores a pointer to its parents, but not vice-versa.
- From any point in the program, symbol table appears to be a stack.
- This is called a **spaghetti stack**.

Type-Checking

- Type errors.
- What are types?
- What is type-checking?
- A simple type system.

5.0 Semantic Analysis

```
class MyClass implements MyInterface {
    string myInteger;

    void doSomething() {
        int[] x;
        x = new string;

        x[5] = myInteger * y;
    }
    void doSomething() {

    }
    int fibonacci(int n) {
        return doSomething() + fibonacci(n - 1);
    }
}
```

Interface not declared

Wrong type

Can't multiply strings

Variable not declared

Can't redefine functions

Can't add void

No main function

What is a Type?

- "The notion varies from language to language.
- The consensus:
  - **A set of values**.
  - **A set of operations on those values"**
- Type errors arise when operations are performed **on values that do not support that operation**.

5.0 Semantic Analysis

Types of Type-Checking

- **Static type checking.**
  - Analyze the program during compile-time to prove the absence of type errors.
  - Never let bad things happen at runtime.
- **Dynamic type checking.**
  - Check operations at runtime before performing them.
  - More precise than static type checking, but usually less efficient.
  - (Why?)
- **No type checking.**
  - Throw caution to the wind!

Type Systems

- The rules governing permissible operations on types forms a **type system**.
- **Strong type systems** are systems that never allow for a type error.
  - Java, Python, JavaScript, LISP, Haskell, etc.
- **Weak type systems** can allow type errors at runtime.
  - C, C++

Typing in Decaf

- Decaf is typed **statically** and **weakly**:
  - Type-checking occurs at compile-time.
  - Runtime errors like dereferencing `null` or an invalid object are allowed.
- Decaf uses **class-based inheritance**.
- Decaf distinguishes primitive types and classes.

Static Typing in Decaf

- Static type checking in Decaf consists of two separate processes:
  - Inferring the type of each expression from the types of its components.
  - Confirming that the types of expressions in certain contexts matches what is expected.
- Logically two steps, but you will probably combine into one pass.

An Example

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Well-typed expression with wrong type.

An Example

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Expression with type error

Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.

Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.

Inferring Expression Types

- How do we determine the type of an expression?
- Think of process as **logical inference**.

Type Checking as Proofs

- We can think of typing checking as proving claims about the types of expressions.
- We begin with a set of axioms, then apply our inference rules to determine the types of expressions.
- Many type systems can be thought of a proof systems.

Sample Inference Rules

- "If **x** is an identifier that refers to an object of type **t**, the expression **x** has type **t**."
- "If **e** is an integer constant, **e** has type `int`."
- "If the operands $\mathbf{e}_1$ and $\mathbf{e}_2$ of $\mathbf{e}_1 + \mathbf{e}_2$ are known to have types `int` and `int`, then $\mathbf{e}_1 + \mathbf{e}_2$ has type `int`."

5.0 Semantic Analysis

Formalizing our Notation

- We will encode our axioms and inference rules using this syntax:

$$\frac{\text{Preconditions}}{\text{Postconditions}}$$

- This is read "if *preconditions* are true, we can infer *postconditions*."

Examples of Formal Notation

$$\frac{\mathbf{A} \rightarrow \mathtt{t}\boldsymbol{\omega} \text{ is a production.}}{\mathtt{t} \in \text{FIRST}(\mathbf{A})}$$

$$\frac{\mathbf{A} \rightarrow \boldsymbol{\varepsilon} \text{ is a production.}}{\boldsymbol{\varepsilon} \in \text{FIRST}(\mathbf{A})}$$

$$\frac{\begin{array}{c}\mathbf{A} \rightarrow \boldsymbol{\omega} \text{ is a production.} \\ \mathtt{t} \in \text{FIRST*}(\boldsymbol{\omega})\end{array}}{\mathtt{t} \in \text{FIRST}(\mathbf{A})}$$

$$\frac{\begin{array}{c}\mathbf{A} \rightarrow \boldsymbol{\omega} \text{ is a production.} \\ \boldsymbol{\varepsilon} \in \text{FIRST*}(\boldsymbol{\omega})\end{array}}{\boldsymbol{\varepsilon} \in \text{FIRST}(\mathbf{A})}$$

Formal Notation for Type Systems

- We write

$$\vdash \mathbf{e} : \mathbf{T}$$

  if the expression **e** has type **T**.
- The symbol $\vdash$ means "we can infer..."

Our Starting Axioms

$$\frac{}{\vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}}$$

Some Simple Inference Rules

$$\frac{i \text{ is an integer constant}}{\vdash i : \texttt{int}}$$

$$\frac{s \text{ is a string constant}}{\vdash s : \texttt{string}}$$

$$\frac{d \text{ is a double constant}}{\vdash d : \texttt{double}}$$

More Complex Inference Rules

$$\frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\vdash e_1 : \texttt{double} \qquad \vdash e_2 : \texttt{double}}{\vdash e_1 + e_2 : \texttt{double}}$$

More Complex Inference Rules

If we can show that e_1 and e_2 have type int…

$$\frac{\vdash e_1 : \texttt{int} \qquad \vdash e_2 : \texttt{int}}{\vdash e_1 + e_2 : \texttt{int}}$$

$$\frac{\vdash e_1 : \texttt{double} \qquad \vdash e_2 : \texttt{double}}{\vdash e_1 + e_2 : \texttt{double}}$$

… then we can show that $e_1 + e_2$ has type int as well

Even More Complex Inference Rules

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 \texttt{ == } e_2 : \texttt{bool}}$$

$$\frac{\vdash e_1 : T \quad \vdash e_2 : T \quad T \text{ is a primitive type}}{\vdash e_1 \texttt{ != } e_2 : \texttt{bool}}$$

Strengthening our Inference Rules

- The facts we're proving have no *context*.
- We need to strengthen our inference rules to remember under what circumstances the results are valid.

5.0 Semantic Analysis

Adding Scope

- We write

$$\mathbf{S} \vdash \mathbf{e} : \mathbf{T}$$

  if, in scope **S**, expression **e** has type **T**.
- Types are now proven relative to the scope they are in.

Old Rules Revisited

$$\frac{}{S \vdash \texttt{true} : \texttt{bool}}$$

$$\frac{}{S \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{i \text{ is an integer constant}}{S \vdash i : \texttt{int}}$$

$$\frac{s \text{ is a string constant}}{S \vdash s : \texttt{string}}$$

$$\frac{d \text{ is a double constant}}{S \vdash d : \texttt{double}}$$

$$\frac{S \vdash e_1 : \texttt{double} \quad S \vdash e_2 : \texttt{double}}{S \vdash e_1 + e_2 : \texttt{double}}$$

$$\frac{S \vdash e_1 : \texttt{int} \quad S \vdash e_2 : \texttt{int}}{S \vdash e_1 + e_2 : \texttt{int}}$$

A Correct Rule

$$\frac{\begin{array}{c} x \text{ is an identifier.} \\ \textbf{\textit{x} is a variable in scope S} \text{ with type T.} \end{array}}{S \vdash x : T}$$

Rules for Functions

f is an identifier.

$$\frac{}{S \vdash f(e_1, \ldots, e_n) \ : ??}$$

Rules for Functions

f is an identifier.
f is a non-member function in scope S.

$S \vdash f(e_1, \ldots, e_n) : ??$

Rules for Functions

f is an identifier.
f is a non-member function in scope S.
f has type $(T_1, \ldots, T_n) \rightarrow U$

$$\frac{}{S \vdash f(e_1, \ldots, e_n) \ : ??}$$

Rules for Functions

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : T_i \text{ for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) : ??}$$

Rules for Functions

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : T_i \ \text{ for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) \ : U}$$

Rules for Arrays

$$\frac{S \vdash \mathbf{e}_1 : \mathrm{T}\,\mathbf{[\,]} \qquad S \vdash \mathbf{e}_2 : \mathbf{int}}{S \vdash \mathrm{e}_1[\mathrm{e}_2] : \mathrm{T}}$$

Rule for Assignment

$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T}{S \vdash e_1 = e_2 : T}$$

Typing with Classes

- How do we factor inheritance into our inference rules?
- We need to consider the shape of class hierarchies.

Rule for Assignment

$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T}{S \vdash e_1 = e_2 : T}$$

If **`Derived`** extends **`Base`**, will this rule work for this code?

```
Base    myBase;
Derived myDerived;

myBase = myDerived;
```

Single Inheritance

Multiple Inheritance

Properties of Inheritance Structures

- Any type is convertible to itself. (**reflexivity**)
- If A is convertible to B and B is convertible to C, then A is convertible to C. (**transitivity**)
- If A is convertible to B and B is convertible to A, then A and B are the same type. (**antisymmetry**)
- This defines a **partial order** over types.

Types and Partial Orders

- We say that $A \leq B$ if A is convertible to B.
- We have that
  - $A \leq A$
  - $A \leq B$ and $B \leq C$ implies $A \leq C$
  - $A \leq B$ and $B \leq A$ implies $A = B$

Updated Rule for Assignment

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_2 \leq T_1}{S \vdash e_1 = e_2 : T_1}$$

Can we do better than this?

Updated Rule for Assignment

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_2 \leq T_1}{S \vdash e_1 = e_2 : \mathbf{T_2}}$$

Updated Rule for Assignment

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_2 \leq T_1}{S \vdash e_1 = e_2 : \mathbf{T_2}}$$

Updated Rule for Comparisons

$$\frac{\begin{array}{c} S \vdash e_1 : T \\ S \vdash e_2 : T \\ T \text{ is a primitive type} \end{array}}{S \vdash e_1 \texttt{ == } e_2 : \texttt{bool}}$$

Updated Rule for Comparisons

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 == e_2 : \texttt{bool}}$$

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \text{ and } T_2 \text{ are of class type.} \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 == e_2 : \texttt{bool}}$$

Updated Rule for Comparisons

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 \texttt{ == } e_2 : \texttt{bool}}$$

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \text{ and } T_2 \text{ are of class type.} \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 \texttt{ == } e_2 : \texttt{bool}}$$

Updated Rule for Comparisons

Can we unify these rules?

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 == e_2 : \texttt{bool}}$$

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \text{ and } T_2 \text{ are of class type.} \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 == e_2 : \texttt{bool}}$$

The Shape of Types

Extending Convertibility

- If A is a primitive or array type, A is only convertible to itself.
- More formally, if A and B are types and A is a primitive or array type:
  - $A \leq B$ implies $A = B$
  - $B \leq A$ implies $A = B$

Updated Rule for Comparisons

$$\frac{S \vdash e_1 : T \quad S \vdash e_2 : T \quad T \text{ is a primitive type}}{S \vdash e_1 == e_2 : \texttt{bool}}$$

$$\frac{S \vdash e_1 : T_1 \quad S \vdash e_2 : T_2 \quad T_1 \text{ and } T_2 \text{ are of class type.} \quad T_1 \leq T_2 \text{ or } T_2 \leq T_1}{S \vdash e_1 == e_2 : \texttt{bool}}$$

$$S \vdash e_1 : T_1$$
$$S \vdash e_2 : T_2$$
$$T_1 \leq T_2 \text{ or } T_2 \leq T_1$$

Updated Rule for Function Calls

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : R_i \ \text{ for } 1 \leq i \leq n \\ R_i \leq T_i \ \text{ for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) \ : U}$$

A Tricky Case

$$\frac{}{S \vdash \texttt{null} : ??}$$

Engine
CarEngine
DieselEngine
bool
string
int
double
Array Types
DieselCarEngine
null Type

Handling `null`

- Define a new type corresponding to the type of the literal `null`; call it "**`null` type**."
- Define `null` type ≤ A for any class type A.
- The `null` type is (typically) not accessible to programmers; it's only used internally.
- Many programming languages have types like these.

A Tricky Case

$$\frac{}{S \vdash \texttt{null} : \texttt{null}\ \text{type}}$$

Object-Oriented Considerations

$$\frac{\text{S is in scope of class T.}}{S \vdash \texttt{this} : T}$$

$$\frac{\text{T is a class type.}}{S \vdash \texttt{new}\ T : T} \qquad \frac{S \vdash e : \texttt{int}}{S \vdash \texttt{NewArray}(e, T) : T\texttt{[]}}$$

Using our Type Proofs

- We can now prove the types of various expressions.
- How do we check...
  - ... that **`if`** statements have well-formed conditional expressions?
  - ... that **`return`** statements actually return the right type of value?
- Use another proof system!

5.0 Semantic Analysis

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }
```

```
while (numBitsSet(x + 5) <= 10) {

    if (1.0 + 4.0) {
        /* … */
    }

    while (5 == null) {
        /* … */
    }

}
```

Expression with type error

Proofs of Structural Soundness

- Idea: extend our proof system to statements to confirm that they are well-formed.
- We say that

$$S \vdash \mathrm{WF}(\mathit{stmt})$$

  if the statement *stmt* is **well-formed** in scope S.
- The type system is satisfied if for every function *f* with body B in scope S, we can show $S \vdash \mathrm{WF}(B)$.

A Simple Well-Formedness Rule

$$\frac{S \vdash expr : T}{S \vdash WF(expr;)}$$

If we can assign a valid type to an expression in scope S…

… then it is a valid statement in scope S.

A More Complex Rule

$$\frac{S \vdash WF(stmt_1) \qquad S \vdash WF(stmt_2)}{S \vdash WF(stmt_1\ stmt_2)}$$

A Rule for Loops

$$\frac{\begin{array}{c} S \vdash \mathit{expr} : \texttt{bool} \\ \text{S' is the scope inside the loop.} \\ S' \vdash \mathrm{WF}(\mathit{stmt}) \end{array}}{S \vdash \mathrm{WF}(\texttt{while (}\mathit{expr}\texttt{)}\ \mathit{stmt})}$$

Rules for **`return`**

$$\frac{\begin{array}{c}\text{S is in a function returning T}\\ S \vdash \mathit{expr} : T' \\ T' \leq T\end{array}}{S \vdash \mathrm{WF}(\texttt{return}\ \mathit{expr}\texttt{;})}$$

$$\frac{\text{S is in a function returning } \texttt{void}}{S \vdash \mathrm{WF}(\texttt{return;})}$$

Checking Well-Formedness

- Recursively walk the AST.
- For each statement:
  - Typecheck any subexpressions it contains.
    - Report errors if **no** type can be assigned.
    - Report errors if the **wrong** type is assigned.
  - Typecheck child statements.
  - Check the overall correctness.

Run-Time Environments

Where we are:

- We have covered the front-end phases
  - Lexical analysis
  - Parsing
  - Semantic analysis
- The front-end phases enforce the definition of the language (syntax)
- It also builds data structures for code generation
- The output of the front end are legal programs
- Next are the back-end phases
  - Optimization
  - Code generation
- We'll do code generation first . . .

- In the back-end phases we are **no longer looking for errors.**
- We are **ready to perform code generation**
- Before discussing code generation**, we need to understand what we are trying to generate**
- There are a number of standard techniques for **structuring executable code** that are widely used

Outline

- Management of run-time resources
- Correspondence between
  - static (compile-time) and
  - dynamic (run-time) structures
- Storage organization

Run-time Resources

- Execution of a program is initially under the control of the operating system
- When a program is invoked:
  - The OS allocates space for the program
  - The code is loaded into part of the space
  - The OS jumps to the entry point (i.e., “main”)

Memory Layout

Notes

- By tradition, pictures of machine organization have:
  - Low address at the top
  - High address at the bottom
  - Lines delimiting areas for different kinds of data
- These pictures are simplifications
  - E.g., not all memory need be contiguous

What is Other Space?

- Holds all data for the program
- Other Space = Data Space
- Compiler is responsible for:
  - Generating code
  - Orchestrating use of the data area

Code Generation Goals

- Two goals:
  - Correctness
  - Speed
- Most complications in code generation come from trying to be fast as well as correct

Assumptions about Execution

1. Execution is sequential; control moves from one point in a program to another in a well-defined order
2. When a procedure is called, control eventually returns to the point immediately after the call

Activations

- An invocation of procedure P is an activation of P
- The life time of an activation of P is
  - All the steps to execute P
  - Including all the steps in procedures P calls

Lifetimes of Variables

- The lifetime of a variable x is the portion of execution in which x is defined
- Note that
  - Lifetime is a dynamic (run-time) concept
  - Scope is a static concept

Activation Trees

- Assumption (2) requires that when P calls Q, then Q returns before P does
- Lifetimes of procedure activations are properly nested
- Activation lifetimes can be depicted as a tree

Example

```
Class Main {
   g() : Int { 1 };
   f():  Int { g() };
   main(): Int {{ g(); f(); }};
}
```

A More Complex example: recursive

```
class Main{
    int g(){ ….}
    int f( int x){
                    if(x==0) then return g();
                     else return f(x-1);
     }
int main() { return f(3);}
```

What is the activation tree for this program?

Notes

- The activation tree depends on run-time behavior
- The activation tree may be different for every program input
- Since activations are properly nested, a stack can track currently active procedures

Revised Memory Layout

Activation Records

- The information needed to manage one procedure activation is called an activation record (AR) or frame.

- If procedure F calls G, then G's activation record contains a mix of info about F and G.

What is in G's AR when F calls G?

- F is "suspended" until G completes, at which point F resumes.
- G's AR contains information needed to resume execution of F.
- G's AR may also contain:
  - G's return value (needed by F)
  - Actual parameters to G(supplied by F)
  - Space for G's local variables

The Contents of a Typical AR for G

- Space for G's return value
- Actual parameters
- Pointer to the previous activation record
  - The control link; points to AR of caller of G
- Machine status prior to calling G
  - Contents of registers & program counter
  - Local variables
- Other temporary values

```
class Main{
    int g(){ ….}
    int f( int x){
                    if(x==0) then return g();
                     else return f(x-1)  (**);
     }
int main() { return f(3)  (*);}
```

Stack after two calls to f

Notes

- Main has no argument or local variables and its result is never used; its AR is uninteresting
- (*) and (**) are return addresses of the invocations of f
- The return address is where execution resumes after a procedure call finishes
- This is only one of many possible AR designs
  - Would also work for C, Pascal, FORTRAN, etc.

The Main Point

- The compiler must determine, at compile-time, the layout of activation records and generate code that correctly accesses locations in the activation record
- Thus, the AR layout and the code generator must be designed together!

Discussion

- The advantage of **placing the return value 1st in a frame** is that the caller can find it at a fixed offset from its own frame
- There is **nothing magic** about this organization
  - Can rearrange order of frame elements
  - Can divide caller/callee responsibilities differently
  - An organization is better if it **improves execution speed or simplifies code generation**

- Real compilers hold as much of the frame as possible in registers
  - Especially the method result and arguments

Global Variables

- All references to a global variable point to the same object
  - Can’t store a global in an activation record
- Globals are assigned a fixed address once
  - Variables with fixed address are “statically allocated”
- Depending on the language, there may be other statically allocated values

Memory Layout with Static Data

Heap Storage

- A value that outlives the procedure that creates it cannot be kept in the AR

  method foo() { new Bar }

- The Bar value must survive deallocation of foo's AR
- Languages with dynamically allocated data use a heap to store dynamic data

Notes

- The code area contains object code
  - For most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- Heap contains all other data
  - In C, heap is managed by malloc and free

- The code area contains object code
  - For most languages, fixed size and read only
- The static area contains data (not code) with fixed addresses (e.g., global data)
  - Fixed size, may be readable or writable
- The stack contains an AR for each currently active procedure
  - Each AR usually fixed size, contains locals
- Heap contains all other data
  - In C, heap is managed by *malloc* and *free*

Notes (Cont.)

- Both the heap and the stack grow
- Must take care that they don’t grow into each other
- Solution: start heap and stack at opposite ends of memory and let them grow towards each other

Memory Layout with Heap

Data Layout

- Low-level details of machine architecture are important in laying out data for correct code and maximum performance
- Chief among these concerns is alignment

Alignment

- Most modern machines are (still) 32 bit
  - 8 bits in a byte
  - 4 bytes in a word
  - Machines are either byte or word addressable
- Data is word alignedif it begins at a word boundary
- Most machines have some alignment restrictions
  - Or performance penalties for poor alignment

Alignment

- Example: A string

"Hello"

Takes 5 characters (without a terminating \0)

- To word align next datum, add 3 "padding" characters to the string
- The padding is not part of the string, it's just unused memory

Next Topic: Stack Machines

- A simple evaluation model
- No variables or registers
- A stack of values for intermediate results
- Each instruction:
  - Takes its operands from the top of the stack
  - Removes those operands from the stack
  - Computes the required operation on them
  - Pushes the result on the stack

Example of Stack Machine Operation

- The addition operation on a stack machine

Example of a Stack Machine Program

- Consider two instructions
  - push i- place the integer ion top of the stack
  - add- pop two elements, add them and put the result back on the stack
- A program to compute 7 + 5:

```
push 7
push 5
add
```

Why Use a Stack Machine ?

- Each operation takes operands from the same place and puts results in the same place
- This means a uniform compilation scheme
- And therefore a simpler compiler

Why Use a Stack Machine ?

- Location of the operands is implicit
  - Always on the top of the stack
- No need to specify operands explicitly
- No need to specify the location of the result
- Instruction “add” as opposed to “add r1, r2”

  ⇒Smaller encoding of instructions

  ⇒More compact programs
- This is one reason why Java Bytecodes use a stack evaluation model

Optimizing the Stack Machine

- The add instruction does 3 memory operations
  - Two reads and one write to the stack
  - The top of the stack is frequently accessed
- Idea: keep the top of the stack in a register (called accumulator)
  - Register accesses are faster
- The "add" instruction is now

$$acc \leftarrow acc + top_of_stack$$

  - Only one memory operation!

Stack Machine with Accumulator

- Invariants
  - The result of an expression is in the accumulator
- For op(e1,…,en) push the accumulator on the stack after computing $e_1,\ldots,e_{n-1}$
  - After the operation pops n-1 values
- Expression evaluation preserves the stack

Stack Machine with Accumulator. Example

- Compute 7 + 5 using an accumulator

A Bigger Example: 3 + (7 + 5)

Code	Acc	Stack
acc ←3	3	<init>
push acc	3	3, <init>
acc ←7	7	3, <init>
push acc	7	7, 3, <init>
acc ←5	5	7, 3, <init>
acc ←acc + top_of_stack	12	7, 3, <init>
pop	12	3, <init>
acc ←acc + top_of_stack	15	3, <init>
pop	15	<init>

Notes

- It is very important evaluation of a subexpression preserves the stack
  - Stack before the evaluation of 7 + 5 is 3, <init>
  - Stack after the evaluation of 7 + 5 is 3, <init>
  - The first operand is on top of the stack

Code Generation

7. Code Generation

- We focus on generating code for a stack machine with accumulator
- We want to run the resulting code on a real machine
  - e.g., the MIPS processor (or simulator)
- We simulate stack machine instructions using MIPS instructions and registers

- The accumulator is kept in MIPS register $a0
- The stack is kept in memory
  - The stack grows towards lower addresses
  - Standard convention on MIPS
- The address of the next location on the stack is kept in MIPS register $sp
  - The top of the stack is at address $sp + 4

Notice that $sp points to the next unused memory, and since the stack grows towards lower addresses the top of the stack is at $sp+4

MIPS architecture

- Prototypical Reduced Instruction Set Computer (RISC)
- Most operations use registers for operands & results
- Use load & store instructions to use values in memory
- 32 general purpose registers (32 bits each)
  - We use $sp, $a0 and $t1 (a temporary register)

- Read the SPIM documentation for details

SPIM is the simulator that we can use to execute our generated code.

5 MIPS instructions that we will need for our first example :

- lw reg_1 offset(reg_2)
  - Load 32-bit word from address reg_2 + offset into reg_1
- add reg_1 reg_2 reg_3
  - $reg_1 \leftarrow reg_2 + reg_3$
- sw reg_1 offset(reg_2)
  - Store 32-bit word in reg_1 at address reg_2 + offset
- addiu reg_1 reg_2 imm
  - $reg_1 \leftarrow reg_2 + imm$
  - "u" means overflow is not checked
- li reg imm
  - $reg \leftarrow imm$

The stack-machine code for 7 + 5 in MIPS:

```
acc ← 7
push acc

acc ← 5
acc ← acc + top_of_stack

pop
```

The stack-machine code for 7 + 5 in MIPS:

acc ← 7	li $a0 7
push acc	sw $a0 0($sp)
	addiu $sp $sp -4
acc ← 5	li $a0 5
acc ← acc + top_of_stack	lw $t1 4($sp)
	add $a0 $a0 $t1
pop	addiu $sp $sp 4

A language with integers and integer operations

$$P \rightarrow D; P \mid D$$

$$D \rightarrow \text{def id(ARGS)} = E;$$

$$\text{ARGS} \rightarrow \text{id, ARGS} \mid \text{id}$$

$$E \rightarrow \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$$
$$\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)$$

- The first function definition f is the entry point
  - The "main" routine

- Program for computing the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
                if x = 2 then 1 else
                   fib(x - 1) + fib(x – 2)
```

- For each expression e we generate MIPS code that:
  - Computes the value of e in $a0
  - Preserves $sp and the contents of the stack

- We define a code generation function cgen(e) whose result is the code generated for e

- The code to evaluate a constant simply copies it into the accumulator:

  cgen(i) = li $a0 i

- This preserves the stack, as required

- Color key:
  - RED: compile time
  - BLUE: run time

```
cgen(e1 + e2) =
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
        cgen(e2)
        lw $t1 4($sp)
        add $a0 $t1 $a0
        addiu $sp $sp 4
```

```
cgen(e1 + e2) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

These two statements store the result of e1 into the stack

```
cgen(e1 + e2) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

These two statements store the result of e1 into the stack

This instruction load the result of e1 into the temporary reg t1

```
cgen(e1 + e2) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

These two statements store the result of e1 into the stack

This instruction load the result of e1 into the temporary reg t1

accu = e1+e1

```
cgen(e1 + e2) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

These two statements store the result of e1 into the stack

This instruction load the result of e1 into the temporary reg t1

accu = e1+e2

pops the stack

```
cgen(e1 + e2) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    add $a0 $t1 $a0
    addiu $sp $sp 4
```

These two statements store the result of e1 into the stack

This instruction load the result of e1 into the temporary reg t1

accu = e1+e1

pops the stack

Actually the code generator creates a file containing the instructions that will be executed at run time.

```
cgen(e1 + e2) =
  cgen(e1)
  print "sw $a0 0($sp)"
  print "addiu $sp $sp -4"
  cgen(e2)
  print "lw $t1 4($sp)"
  print "add $a0 $t1 $a0"
  print "addiu $sp $sp 4"
```

- The code for + is a template with "holes" for code for evaluating e_1 and e_2
- Stack machine code generation is recursive
  - Code for $e_1 + e_2$ is code for e_1 and e_2 glued together
- Code generation can be written as a recursive-descent of the AST
  - At least for expressions

- New instruction: sub reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

```
cgen(e1 - e2) =
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
        cgen(e2)
        lw $t1 4($sp)
        sub $a0 $t1 $a0
        addiu $sp $sp 4
```

- New instruction: sub reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

```
cgen(e1 - e2) =
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
        cgen(e2)
        lw $t1 4($sp)
        sub $a0 $t1 $a0
        addiu $sp $sp 4
```

What is the difference between this and the code for e1+e2?

- New instruction: sub reg_1 reg_2 reg_3
  - Implements $reg_1 \leftarrow reg_2 - reg_3$

```
cgen(e1 - e2) =
        cgen(e1)
        sw $a0 0($sp)
        addiu $sp $sp -4
        cgen(e2)
        lw $t1 4($sp)
        sub $a0 $t1 $a0
        addiu $sp $sp 4
```

This difference is only in this instruction (sub instead of add)

To be able to generate code for **if expressions** we need two more MIPS instructions:

- New instruction: beq reg_1 reg_2 label
  - Branch to label if $reg_1 = reg_2$
- New instruction: b label
  - Unconditional jump to label

```
cgen(if e1 = e2 then e3 else e4) =
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen(e2)
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true_branch
false_branch:
  cgen(e4)
  b end_if
true_branch:
  cgen(e3)
end_if:
```

- Code for function calls and function definitions depends on the layout of the AR

- A very simple AR suffices for this language:
  - The result is always in the accumulator
    - No need to store the result in the AR
  - The activation record holds actual parameters
    - For $f(x_1,\ldots,x_n)$ push $x_n,\ldots,x_1$ on the stack
    - These are the only variables in this language

- The stack discipline guarantees that on function exit $sp is the same as it was on function entry
  - No need for a control link
- We need the return address
- A pointer to the current activation is useful
  - This pointer lives in register $fp (frame pointer)

- Summary: For this language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices

- Picture: Consider a call to f(x,y), the AR is:

- The calling sequence is the instructions (of both caller and callee) to set up a function invocation

- New instruction: jal label
  - Jump to label, save address of next instruction in $ra
  - On other architectures the return address is stored on the stack by the "call" instruction

The code that will execute on the Caller Side

```
cgen(f(e1,....en)) =
    sw $fp 0($sp)
    addiu $sp $sp -4
    cgen(en)
    sw $a0 0($sp)
    addiu $sp $sp -4
    ...
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    jal f_entry
```

The caller saves its value of the frame pointer

Then it saves the actual parameters in reverse order

Finally the caller saves the return address in register $ra

The AR so far is 4*n+4 bytes long

The code that will execute on the Callee Side

- New instruction: jr reg
  - Jump to address in register reg

cgen(def f(x_1,…,x_n) = e) =

```
Entry: move $fp $sp
       sw $ra 0($sp)
       addiu $sp $sp -4
       cgen(e)
       lw $ra 4($sp)
       addiu $sp $sp z
       lw $fp 0($sp)
       jr $ra
```

- New instruction: jr reg
  - Jump to address in register reg

cgen(def f(x_1,…,x_n) = e) =

```
Entry: move $fp $sp
       sw $ra 0($sp)
       addiu $sp $sp -4
       cgen(e)
       lw $ra 4($sp)
       addiu $sp $sp z
       lw $fp 0($sp)
       jr $ra
```

- Note: The frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- z = 4*n + 8

Two words: the return address and the old frame address

Before call
FP
SP
On entry
FP
old fp
y
x
SP
Before exit
old fp
y
x
FP
return
SP
After call
FP
SP

Generating Code for Reference Variables: i.e. generate code that determines their place in the activation record.

- Variable references are the last construct
- The "variables" of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller
- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp

- Solution: use a frame pointer
  - Always points to the return address on the stack
  - Since it does not move it can be used to find the variables
- Let x_i be the ith ($i = 1,\ldots,n$) formal parameter of the function for which code is being generated

$$cgen(x_i) = \text{lw } \$a0\ z(\$fp) \qquad (\ z = 4*i\ )$$

- Example: For a function def f(x,y) = e the activation and frame pointer are set up as follows:

- X is at fp + 4
- Y is at fp + 8

Code Generation Example

def sumto(X) = if x=0 then 0 else x+sumto(x-1)

Sumto_entry:

```
 move    $fp    $sp          % fp points to AR of sumto
sw       $ra    0($sp)
addiu    $sp    $sp    -4
lw       $a0    4($fp)       //load  x
sw       $a0    0($sp)
addiu    $sp    $sp   -4
li       $a0    0            //code for 0
lw       $t1    4($sp)       //pop x
addiu    $sp    $sp    4
beq      $a0    $t1   true1  //goto then part
```

false1:

false1:

```
        lw      $a0     4($fp)      % x
        sw      $a0     0($sp)
        addiu   $sp     $sp  -4
% call sumto(x-1) here we insert what happens at the caller side
% store old fp on the new activation record
        sw      $fp     0($sp)
        addiu   $sp     $sp   -4
% compute x-1 and store it in the new AR
        lw      $a0     4($fp)
        sw      $a0     0($sp)
        addiu   $sp     $sp  -4
        li      $a0     1           //code for 1
        lw      $t1     4($sp)      // x
        sub     $a0     $t1   $a0
        addiu   $sp     0($sp)      //pop stack
%store x-1 in the new AR
        sw      $a0     0($sp)
```

```
% jump to sumto(x-1)
        jal     Sumto_entry
% add X
        lw      $t1     4($sp)      % x was stored in the stack
                                    because it is part of the
                                    expression x+sumto(x-1)
        add     $a0     $t1     $a0
        addiu   $sp     $sp     4   %pop x
        b       endif
% the true part    0
true1:
        li      $a0     0
% function ends so we return
endif:  lw      $ra     4($sp)
% pop AR
        addiu   $sp     $sp     12
        lw      $fp     0($sp)
% jump back to the caller
        jr      $ra
```

Two Observations:

1. Although the code is correct it is extremely inefficient, notice that we load x form the stack (memory op) then immediately after that we store it back in the stack.
2. Temporary values are stored in the AR; a better alternative might be to store them in temporary registers.

Summary

- The activation record must be designed together with the code generator
- Code generation can be done by recursive traversal of the AST
- The stack model simplifies things.
- Production compilers do different things
  - Emphasis is on keeping values (esp. current stack frame) in registers
  - Intermediate results are laid out in the AR, not pushed and popped from the stack

An Improvement

- Idea: Keep temporaries in the AR
- The code generator must assign a fixed location in the AR for each temporary (this saves the need to keep pushing an popping them).

Example

```
def fib(x) = if x = 1 then 0 else
        if x = 2 then 1 else
            fib(x - 1) + fib(x – 2)
```

- What intermediate values are placed on the stack?
- How many slots are needed in the AR to hold these values? i.e. how many intermediate values are needed?

How Many Temporaries?

- Let NT(e) = # of temps needed to evaluate e
- NT(e_1+ e_2)
  - Needs at least as many temporaries as NT(e_1)
  - Needs at least as many temporaries as NT(e_2) + 1
- Space used for temporaries in e_1 can be reused for temporaries in e_2

The Equations

- $NT(e_1 + e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(e_1 - e_2) = \max(NT(e_1), 1 + NT(e_2))$
- $NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) = \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4))$
- $NT(id(e_1,\ldots,e_n) = \max(NT(e_1),\ldots,NT(e_n))$
- $NT(int) = 0$
- $NT(id) = 0$

- What is NT(…code for fib…)?

- The answer is 2

def fib(x) = if x = 1 then 0 else

if x = 2 then 1 else

fib(x - 1) + fib(x – 2)

The Revised AR

- For a function definition $f(x_1,\ldots,x_n) = e$ the AR has $2 + n + NT(e)$ elements
  - Return address
  - Frame pointer
  - n arguments
  - NT(e) locations for intermediate results

Picture

Old FP
x_n
. . .
x_1
Return Addr.
Temp NT(e)
. . .
Temp 1

Revised Code Generation

- Code generation must know how many temporaries are in use at each point
- Add a new argument to code generation: the position of the next available temporary

Code Generation for + (original)

cgen(e_1+ e_2) =

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

Code Generation for + (revised)

cgen(e_1+ e_2, nt) =

cgen(e_1, nt)

sw $a0 nt($fp)

cgen(e_2, nt + 4)

lw $t1 nt($fp)

add $a0 $t1 $a0

Notes

- Two sentences shorter and substantially more efficient.
- The temporary area is used like a small, fixed-size stack

Key solution

King Saud University
College of Computer and Information Sciences
Computer Science Department

Final Exam

Academic Year: 2015/2016
Second Semester

BSc Program
Course Name/No. : Programming Language Compilation / CS340

Exam Date: 19 /5/2016:

Total Number of Pages: 9 pages (including this cover page)

Student Name	
Student ID.	Seat Number

Question No.	Total Mark	Student Mark
1.	**5**	
2.	**7**	
3.	**6**	
4.	**9**	
5.	**6**	
6.	**7**	
Total	**40**	

Q1) A) Write a regular expression that describes the language of all integers that consist of at most 3 digits with an optional sign (+ or -). (2 marks)

digit = '0' + '1' + '2' + --- + '9'

(+ + - + ε)(digit + digit digit + digit digit digit)

B) Design a DFA that accepts an integers that if it consist of at most 3 digits with an optional sign (+ or -). (2 marks)

c) Give an example of a language that is irregular (not regular) (1 mark)

$= \{ (^n)^n \}$

Q2) Consider the following regular expression

(A*+B*)1

A) Convert the regular expression you wrote above into an NFA. (2 marks)

B) Convert the NFA you wrote in B into a DFA. (2 marks)

C) Represent the DFA using a table (1 mark)

	A	B	1
1	2		4
2	2		3
3			
4			

D) Write an algorithm (pseudo code) that a string as input and displays "yes" if a string belongs to the language **(A*+B*)1**; otherwise it displays "no". (2 marks)

```
input = ∅
state = ∅
while (moreInput)
    state = T[state, input++];
if (state = accept state)
    print "yes"
else
    print "no"
```

Q3) Consider the CFG

E → E+E|E*E|int|id

a) Give an example that illustrates that the above grammar is ambiguous. (2 marks)

int + int * int has more than one parse tree

b) Rewrite the grammar so that * has higher precedence than +. (2 marks)

E → E+E' | E'

E' → E'*E' | int | id

int + int * int

C) Assuming that ambiguity is fine, rewrite the given grammar so that we can write a recursive descent parser for it. (2 marks)

E → int+E | id+E | int*E | id*E

Q4) Consider the following grammar

E→TX
T→(E)|int Y | ε
X→+E| ε
Y→*T| ε

A) Find the first set and follow set for each terminal and non-terminal symbol used in the grammar. (3 marks)

follow (T) = first (X) ∪ follow(E)
follow (E) = { $,) } ∪ ~~follow(T)~~ ∪ follow (X)
~~follow(T)~~ = follow(E) = follow(X) = { $,) }

Symbol	First set	Follow set
T	{(, int, ε}	{+, $,)}
E	{(, int, +, ε}	{$,), +}
X	{+, ε}	{$,), +}
Y	{*, ε}	{$,), +}
(	{(}	{(, int, +, $} = first(E)
)	{)}	follow (T) = {+, $,)}
int	{int}	{*, +, $,)} = first(Y) ∪ follow(T)
+	{+}	first (E) = {(, int, +, $,)}
*	{*}	first(T) ∪ follow(Y) = {(, int, $,), +}

B) Draw the LL(1) parsing table. (2 marks)

	(	)	int	+	*	$
E	TX	TX	TX	TX		TX
T	(E)	ε	int Y	ε		ε
X		ε		+E		ε
Y		ε		ε	*T	ε

E) Write down the SLR(1) parsing algorithm. (2 marks)

c) Assuming dynamic type checking, rewrite the above rules so that they work for reference types. (2 marks)

$$\frac{S \vdash e_1 : T \qquad S \vdash e_2 : T}{S \vdash e_1 = e_2 : T}$$

if e_1 and e_2 are of the same type, T, then the assignment exp $e_1 = e_2$ has also type T

Q6) A) Write cgen for the expression e1+e2 (3 marks)

```
cgen( e1+e2)
cgen(e1)
sw $a0 0($sp)
addiu $sp -4
cgen( e2)
lw $t1 4($sp)
add $a0 $a0 $t1
addiu $sp $sp 4
```

Q5) a) Why is "null" causes a problem for type systems? And how do they deal with it?

(2 marks)

we declare a null type at the bottom of the inheritance hierarchy

b) In your own words, describe what the following rule means. (2 marks)

1.

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope } S. \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : T_i \text{ for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) : U}$$

if the arguments $T_1, \ldots, T_n$ have type $e_1, \ldots, e_n$ then the function invocation $f(e_1, \ldots, e_n)$ has type U

B) Write code that cgen generates for the expression 5+7. (4 marks)

```
li $a0 5
sw $a0 0($sp)
addiu $sp 4
li $a0 7
lw $t1 4($sp)
add $a0 $a0 $t1
addiu $sp $sp 4
```

Some MIPS instruction:

- lw reg1 offset(reg2) % load 32-bit word from address reg2+offset into reg1
- add reg1 reg2 reg3 % reg1 = reg2+reg3
- sw reg1 offset(reg2) % Store 32-bit word in reg1 at address reg2 + offset
- addiu reg1 reg2 imm % reg1=reg2+imm
- li reg imm % reg=imm
- beq reg1 reg2 label % branch to label if reg1=reg2
- ble reg1 reg2 label % branch if reg1<=reg2
- b label % unconditional jump to label

Register names:

Recall that in MIPS the accumulator is $a0, the Stack pointer is $sp, and the temporary register is $t1

Key Solution

King Saud University
College of Computer and Information Sciences
Computer Science Department

Final Exam
Academic Year: 2015/2016
First Semester

BSc Program
Course Name/No. : Programming Language Compilation / CS340

Exam Date: 30 /12/2015:

Total Number of Pages: 8 pages (including this cover page)

Student Name	
Student ID.	

Question No.	Total Mark	Student Mark
1.	6	
2.	7	
3.	8	
4.	8	
5.	4	
6.	7	
Total	40	

Q1) Consider the following NFA

A) Describe the language of this NFA using a regular expression. (2 marks)

$(a+b)^* abbb$

B) Convert the above NFA into a DFA. (4 marks)

Q2) Consider the CFG

E → E-E | E+E
E → int | id

a) Rewrite the CFG in such a way that allows you to write a recursive descent parser for it. (2 marks)

Eliminate left recursion

E → int E' | id E' | int | id

E' → +E | -E

b) Write a recursive descent parser in Java (or as s pseudo code) for the grammar you wrote in above (in A). (3 marks)

```
bool term(Token tok){
        return next++tok;
}
bool E1(){return term(int) && EP();}
bool E2(){return term(id) && EP();}
bool E3(){return term(id);}
bool E4(){return term(int);}
bool E(){Token save=next; return(next=save, E1()
                                 || next=save, E2()
                                 || next=save, E3()
                                 || next=save, E4() }
bool EP1(){ return term + && E();}
bool EP2(){return term(-) && E();}
bool E'(){Token save=next; return(next=save, EP1() ||
                                  (next=save, EP2
```

c) Rewrite the grammar in such a way that makes an LL(1) parser possible for the grammar. (2 marks)

E → int X | id X
X → E' | ε
E' → +E | -E

Q3) A) What is the main reason that makes parsers unable to detect all errors? Give examples of 4 errors that cannot be detected by a parser. (3 marks)

Reason:

because CFG cannot capture all prog. lang. constructs or describe

Examples: 1)

using undeclared variables

2)

data type mismatch

3)

using a void function in expressions

4) A class that does not correctly implement an interface.

b) In your own words, describe what each of the following rules mean. (2+1 marks)

1.

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : T_i \text{ for } 1 \leq i \leq n \end{array}}{S \vdash f(e_1, \ldots, e_n) : U}$$

if each of the arguments $e_1, e_2, \ldots, e_n$ has type $T_1, T_2, \ldots T_n$, then the return type of f is U

2.

$$\frac{\begin{array}{c} S \vdash e_1 : T \\ S \vdash e_2 : T \end{array}}{S \vdash e_1 = e_2 : T}$$

if e_1 has type T and e_2 also has type T, then $e_1 = e_2$ has type T

B) Assuming dynamic type checking, rewrite the above rules so that they work for reference types. (2 marks)

1.

f is an id
f is a function in scope S
f has type $(T_1, \ldots, T_n) \rightarrow U$
$S \vdash e_i : R_i$ and $R_i \leq T_i$ for $1 \leq i \leq n$

$S \vdash f(e_1, \ldots, e_n) : U$

2.

$S \vdash e_1 : T_1$
$S \rightarrow e_2 : T_2$
$T_2 \leq T_1$

$S \vdash e_1 = e_2 : T_2$

Q4 Consider the following DFA that recognizes the viable prefixes for the grammar

E→ E'
E→ E+T | T
T→T*F | F
F→(E) | id

A) Construct an LL(1) table for the grammar. (4 marks)

	+	*	(	)	id	$
E'			E		E	
E			T/E+T		T/E+T	
T			F/T*F		F/T*F	
F			(E)		id	

B) Is the grammar an LL(1) grammar? Why or why not? (1 mark)

No, because it has a multiply defined entry.

c) The following NFA recognizes the viable prefixes for the above grammar. Is the grammar an SLR(1) grammar? Why or why not? (3 marks)

Yes it is because + is not in follow of E' and * is not in the follow of E

Q5) Write cgen for each of the following expressions (Hint you can use the MIPS instructions at the last page). (3+1 marks)

1) If e1=e1 then e3 else e4

```
cgen (if e1=e2 then e3 else e4)
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen(e2)
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true-if
    cgen(e4)
    b endif
  true-if:
    cgen(e3)
  end-if:
```

2) 5

```
li $a0 5
```

Q6) A) Write code that cgen generates for following procedure

def add(x,y)=x+y+2;

```
~~cgen~~ (
f_entry:
  move $fp $sp
  sw $ra 0($sp)
  addiu $sp $sp -4
  lw $a0 4($fp)
  sw $a0 4($sp)
  addiu $sp $sp -4
  lw $a0 8($fp)
  lw $t1 4($sp)
  addiu $sp $sp 4
  add $a0 $t1 $a0
  lw $ra 4($sp)
  addiu $sp $sp 4
  ~~addiu $fp~~
  lw $fp 0($sp)
  jr $ra
```

B) Write the code that cgen generates for the invocation statement

add(4,5)

```
sw $fp 0($sp)
addiu $sp $sp 4
li $a0 4
sw $a0 0($sp)
addiu $sp $sp 4
li $a0 5
sw $a0 0($sp)
addiu $sp $sp 4
jal f_entry
```

Key Solution

King Saud University
College of Computer and Information Sciences
Computer Science Department

Final Exam

Academic Year: 2016/2017
Second Semester

BSc Program
Course Name/No. : Programming Language Compilation / CS340

Exam Date: 17 /5/2017:

Total Number of Pages: 7 pages (including this cover page)

Student Name	
Student ID.	

Exercise No.	**Full Mark**	**Student Mark**
1.	6	
2.	4	
3.	8	
4.	9	
5.	5	
6.	8	
Total	40	

Q1) a) Write regular expressions for a language over the alphabet $\Sigma = \{a,b\}$ that accepts all strings that begin and end with a different character. (2 marks)

$a(a+b)^* b + b(a+b)^* a$

b) Design a DFA, over the alphabet $\Sigma = \{a,b\}$, that accepts a string if contains exactly two **b**'s and one **a**. (2 marks)

c) Convert the regular expression **(aa*)+b** into an NFA (2 marks)

Q2) Consider the following CFG

X → X+X | X*X |int

a) Show that the grammar is ambiguous. (2 marks)

Since int + int + int has the following 2 parse trees, the grammar is ambiguous

b) Rewrite the grammar so that + has left associativity and * has right associativity. (2 marks)

$X \rightarrow X+\text{int} \mid \text{int} * X \mid \text{int}$

Q3) Consider the following grammar over the alphabet ∑ = {u,v,w,x,y,z};

S → UVW

U → u | Wv | ϵ

V → w | xU | ϵ

W → y | z | ϵ

a) Complete the missing entries in the following table that gives the first and follow sets for each terminal and non-terminal symbols: (4 marks)

Symbol	First	Follow
S	{u, y, z, v, w, x, ε}	{$}
U	{u, y, z, v, ε}	{w, x, y, z, $}
V	{w, x, ε}	{y, z, $}
W	{y, z, ε}	{$, v}
u	{u}	{w, x, y, z, $}
v	{v}	{v, x, y, z, $}
w	{w}	{y, z, $}
x	{x}	{u, y, z, v, $}
y	{y}	{$, v}
z	{z}	{$, v}

b) Draw the corresponding LL(1) table. (3 marks)

	u	v	w	x	y	z	$
S	uvw	uvw	uvw	uvw	uvw	uvw	
U	u	wv	ε	ε	wv / ε	wv / ε	ε
V			w	xU	ε	ε	ε
W		ε			y	z	ε

b) Is the grammar an LL(1) grammar? Why? or why not? (1 marks)

it is not because we have a multiply defined entry

Q4) A) Draw an NDF that recognizes all valid items for the grammar (3 marks)

X → X+X | X*X |int

c) Is that above grammar SLR grammar? Why? or why not? (2 marks)

D) Consider the following DFA for viable prefixes of a grammar **(4 marks)**

Assuming that an SLR parser resolves shift-reduce conflicts by choosing to shift, show the operation of such a parser on the input string **{stmt;;}**. Write your answer as a table of a suitable format and make it clear if the statement is accepted or rejected.

Configuration	DFA	Action
1 {stmt;;}$	1	Shift
{1 stmt;;}$	2	Shift
{stmt1;;}$	5	reduce, S->stmt
{S 1;;}$	⋮	⋮

Q5) Consider the following program

```
0:  int x = 137;
1:  int z = 42;
2:  int MyFunction(int x, int y) {
3:    printf("%d, %d, %d\n", x, y, z);
4:    {
5:      int x, z;
6:      x = y;
7:      x = z;
8:      {
9:        int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d, %d, %d\n", x, y, z);
16:   }
17: }
```

1) Mark each variable with the line number it was declared in, by writing above it @ followed by the line number. (3 marks)

2) Draw the symbol table that you have used when you reached line 13. (2 marks)

x	0
z	1
x	2
y	2
x	5
z	5
y	9

Q6) A) Write a code generator for the language constructs If e1=e2 then e3 else e4 (4 marks)

```
cgen(if e1=e2 then e3 else e4)
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
addiu $sp $sp 4
beq $a0 $t1 true_branch
false_branch:
cgen(e4)
b end_if
true_branch:
cgen(e3)
end_if:
```

B) Write the code that would be produced by cgen for the expression **x+5**. You can assume that x is the only variable in the procedure. (4 marks)

```
lw $a0 4($fp)
sw $a0 0($sp)
addiu $sp $sp -4
li $a0 5
lw $t1 (4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

final

Key solution 1st term Form A

King Saud University
College of Computer and Information Sciences
Computer Science Department

Final Exam

Academic Year: 2016/2017
First Semester

Master Program
Course Name/No. : Programming languages and Compilation / CSC 340

Exam Date: 18/1/2017:
Exam Time: 180 minutes: From 8:00 To 11:00 am.

Total Number of Pages: 6 pages (including this cover page)

Student Name	
Student ID.	

Exercise No.	Full Mark	Student Mark
1.	**30**	
2.	**10**	

Total 40

Q1 Answer: Please copy your answer to question 1 to this table.

1.	2.	3.	4.	5.	6.	7.	8.	9.	10
b	c	d	c	c	a	b	a	d	b

11.	12.	13.	14.	15.	16.	17.	18.	19.	20
b	d	c	✓	a	d	d	d	a	b

21.	22.	23.	24.	25.	26.	27.	28.	29.	30
b	a	~~d~~ a or c	b	d	c	a	c	d	a

Q1) Put a circle around the best answer for each of the following then copy your answer to the table on the first page (only the table will be graded). (30 grades)

1. The main advantage of interpreters over compliers is
   a. efficiency
   b. portability of programs
   c. reliability
   d. a+b
2. Java programs are partly compiled and partly interpreted to achieve
   a. portability
   b. efficiency
   c. portability and efficiency
   d. reliability
3. Which of the following programming language features affect the reliability of a programming language
   a. Data types
   b. Exception handling
   c. Orthogonality
   d. All of the above
4. Which of the following programming language features affects the writability of the programming language
   a. Data types
   b. Exception handling
   c. Orthogonality
   d. All of the above
5. Which of the following regular expressions describe the language over the alphabet {a,b} that consists of all strings that contain at least one 'a'.
   a. $b^*aa^*b^*$
   b. $b^*ab^*a^*b^*$
   c. $b^*a(a^*b^*)^*$
   d. none of the above
6. Which of the following regular expressions describe the language over the alphabet {a,b} that consists of all strings that begin and end with the same letter.
   a. a(b*a)*+b(a*b)*
   b. ab*a+ba*b
   c. ab*a*b*a+ba*b*a*b
   d. none of the above
7. Converting an NFA that consists of n states to an equivalent DFA would result in a DFA that contains
   a. exactly 2^n states
   b. at most 2^n states
   c. at least 2^n states
   d. n! states
8. A table that represents an NFA of 3 states and deals with 2 terminal symbols consists of
   a. 3 rows and 3 columns
   b. 3 rows and 2 columns
   c. 2 columns and 3 rows
   d. 5 rows and 2 columns

9. Which of the following methods require eliminating left recursion
   a. Recursive descent parsing
   b. LL(1) parsing
   c. Bottom up parsing
   d. a+b
10. Which of the following methods may require left factoring a grammar
   a. Recursive descent parsing
   b. LL(1) parsing
   c. Bottom up parsing
   d. a+b
11. Which of the following productions assigns left associativity for +
   a. E→ int+E
   b. E→E+int
   c. E→E+E|int
   d. None of the above
12. Which of the following grammars is not ambiguous
   a. E→ id+E | id*E | (E) | id
   b. E→E+E | E * E | (E) | id
   c. E→E+T|T
      T→ T*F | F
      F→ (E) | id
   d. a+c

Question 13-16 are related to the following grammar

E→E+T|T
T→ T*F | F
F→ (E) | id

13. What is follow set of E?
   a. { + }
   b. { +,) }
   c. { $, +,) }
   d. { $, + ,), ε }
14. What is follow set of F?
   a. { + }
   b. { +,) }
   c. { $, +,) }
   d. { $, + ,), ε }
15. What is first set of E?
   a. { (, id }
   b. { (}
   c. {id}
   d. { (, id, ε }
16. In the LL(1) table for the above grammar, the entry that contains E+T is
   a. row E column (
   b. row E column id
   c. row E column $
   d. a+b

The following assumptions are related to questions 17-19

Assume that a grammar contains the production E→T+F|TF,
Assume also that first(T)={id, ε}, fist(F)= { * , ε }, follow(E)={id,$} and follow(T)={+, $}

17. In the corresponding LL(1) table, which entry contains TF
    a. row E, column id
    b. row E, column +
    c. row E, column *
    d. a and b
18. which entry in the LL(1) table contains ε
    a. row E, column id
    b. row E, column +
    c. a and b
    d. none of the above
19. The grammar is not LL(1) grammar because the entry at
    a. row E column id is multiply defined
    b. row E column $ is multiply defined
    c. row E column + is multiply defined
    d. the grammar is LL(1) because no entry is multiply defined
20. Bottom-up parsers are
    a. More efficient than top-down parsers
    b. More general than top-down parsers
    c. Less general than top-down parsers
    d. a and b

The following assumptions are related to questions 21-...

Assuming that the DFA for recognizing the viable prefixes of a grammar contains the items
E→T.+F
T→F.

21. Which types of conflicts does the above grammar contain
    a. Reduce-reduce conflict
    b. Shift reduce conflict
    c. No conflicts
    d. A+b
22. The grammar is not SLR(1) grammar if
    a. follow(E)={+,$}
    b. follow(T)={+}
    c. first(F)={+}
    d. none of the above
23. If the top of the stack contains + and the next input token is +, then the SLR parsing algorithm
    a. pops + from the stack
    b. reports and error
    c. moves to the next input token
    d. a+b

a + c

24. If the top of the stack contains + and the next input token is *, then the SLR parsing algorithm
    a. pops + from the stack
    b. reports and error
    c. moves to the next input token
    d. a+b
25. Which of the following errors are not detected by a parser
    a. Undeclared identifier
    b. Unexpected data type
    c. Identifier declared more than once
    d. All of the above

Consider the following rule to answer questions 27- 29

$$\frac{\begin{array}{c} f \text{ is an identifier.} \\ f \text{ is a non-member function in scope S.} \\ f \text{ has type } (T_1, \ldots, T_n) \rightarrow U \\ S \vdash e_i : R_i \text{ for } 1 \le i \le n \\ R_i \le ?? \text{ for } 1 \le i \le n \end{array}}{S \vdash f(e_1, \ldots, e_n) : ?}$$

26. What should be written in place of the double question marks (??) in the rule to make correct?
    a. U
    b. f
    c. T_i
    d. None of the above
27. What should be written in place of the question mark (?) in the rule to make correct?
    a. U
    b. f
    c. T_i
    d. None of the above
28. Assuming that a primitive type is convertible to itself, then the above inference rule is applicable for
    a. Referenced types
    b. Primitive types
    c. a and b
    d. None of the above
29. In a stack machine, the heap section of the memory is dedicated for
    a. Local variables
    b. Actual arguments
    c. a and b
    d. dynamically allocated objects
30. The memory section allocated to global variables is allocated
    a. Statically
    b. Dynamically
    c. At execution time
    d. None of the above

Q2) a) Write the code generator segment that deals with the if statement (4 grades)

cgen(if e1=e2 then e3 else e4)

```
cgen(if e1=e2 then e3 else e4)
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    addiu $sp $sp 4
    beq $a0 $t1 true-branch
false-branch:
    cgen(e4)
    b end_if
true-branch:
    cgen(e3)
end_if:
```

b) What code will be generated for the following procedure? i.e. what would be produced by the compiler? (6 grades)

def f(x,y) =

if x=y then x else 0

```
f_entry:
    move $fp $sp
    sw $ra 0($sp)
    addiu $sp $sp -4
    // cgen(...)
    lw $a0 4($fp)   // x
    sw $a0 0($sp)
    addiu $sp $sp -4   push x
    lw $a0 8($fp)   // y
    lw $t1 4($sp)
    addiu $sp $sp 4   pop x
    beq $a0 $t1 true-branch
false_branch:
    li $a0 0
    b end_if
true-branch:
    lw $a0 4($fp)
end_if:
    lw $ra 4($sp)
    addiu $sp $sp 16
    lw $fp 0($sp)
    jr $ra
```

Student Name:

Section: time (11-12)

Q1) (4 marks) A) Discuss two features that make Java better with respect to reliability than C++.

1. Java it have indexing and it will check if the index out of bound or not.
in c++ it have Pounter, it allows user to go out of bound.

2

2. Java it will transalit to inter madat langug
c++ it will transalit to machean langug.

2

B) (2 marks) Explain how the implementation of Java makes it better with respect to portability than C++.

Java it will transalit to intermadat langug
Be for transalit to machean langug
that will respect to Protability.
~~it will work w be effecnt work with allot of OS.~~

2

C) (4 marks) Discuss two advantages of C++ compared to Java.

1. ~~easay to learn~~

respect the cost

2. ~~easay to~~ ~~in some caus it~~

~~will be easy to learn~~

easisir to learn.

Q2) (4 marks) a) Assuming that a float number consists of an optional integer part and an optional fraction part, write a regular expression that describes the language of all float numbers. Note that the strings "14.15", ".15" and "15." are all legal but the string "." is not.

dig = [0-9];

$(dig^{+} + \cdot)(dig)(dig^{*} + \cdot)$

b) (4 marks) Write a CFG that describes the language of float numbers as described above.

S → dig . dig | . dig | dig .

C) (2 marks) Consider a language defined over S={0,1} where a string belongs to the language if the number of 1's is twice the number of 0. What would you use describe this language regular expressions or context free grammar? Justify your answer

I will use context free grammar.

because it powre full than regular expressions and context free grammar have memory.

2

Q3) (4 marks) A) Draw an NFA that recognizes the language denoted by the regular expression 1(0+1*).

3

B) (2 marks) Write a regular expression that describes the language recognized by the following NFA.

{000}
{111}
{011}
{01}
{1}

1

(000+111+011+01+1)

0 repeated

Ex.Midterm_1_2016

C) (2 marks) Represent the above NFA using a table.

	0	1	ε
→A	{F}	{B}	∅
B	∅	{C}	{D}
E	{F}	∅	{B, C}
C	∅	{D}	∅
F	{D}	∅	∅
*D	∅	∅	∅

D) (4 marks) Write an algorithm that uses the above table to determine if a given string is legal.

```
i=0;
Stauts=0;
while ( input[i]){
if( Stauts==A && input[i]==0){
```

King Saud University
College of Computer and Information Sciences
Computer Science Department
CSC 340 Mid-1

Key

mid term 2016 - 2017

Student Name: Student Number:

Q1) (14 marks) Put a circle around the symbol of the best answer for each of the following

1) Which of the following features must be available in a programming language to be suitable for writing embedded systems
   a) Provide constructs for low-level control over hardware.
   b) Use floating point representation
   c) Have features to analyze data
   d) All of the above.
2) An orthogonal programming language
   a. Has a relatively small set of primitive constructs that can be combined in a relatively small number of ways to get the desired results.
   b. Has a fewer number of exceptions because every possible combination is legal.
   c. Is good with respect to reliability
   d) All of the above
3) Which of the following features a programming language need to have in order to be good with respect to reliability
   a. Data types
   b. Support for abstraction
   c. Exception handling
   d) all of the above
4) Having a small number of manageable features and constructs, makes a programing language good with respect to
   a. Readability
   b. Writability
   c. Reliability
   d) All of the above
5) A language with too many operators and special symbols is
   a. good with respect to writability
   b. bad with respect to reliability
   c. bad with respect to readability
   d) all of the above
6) Java is partly compiled and partly interpreted (hybrid implementation) makes is
   a) Good for writing portable programs
   b. Good for writing efficient programs
   c. Good for reducing the cost of training
   d. All of the above
7) Classes, inheritance and polymorphism designed in programming languages as a result of
   a. The Von-Neumann architecture
   b) Programming methodologies
   c. People preferences
   d. None of the above

Q 2) Consider the following DFA over the alphabet Σ ={a,b}

a) (2 marks) In your own words describe the language accepted by the above DFA.

accepts a string if it has 3 consecutive a's

b) (4 marks) Write a regular expression that describes the language accepted by the above DFA. $(a+b)^* aaa (a+b)^*$

or

$b^* aaa (a|b)^* | b^*(ab^*|aab^*)^* aaa(ab)^*$

c) (2 marks) Represent the DFA using a table.

$b^* aaa(a|b)^* | b^*(ab^*|aab^*)^* aaa(a|b)^*$

	q_0	q_1
q_0	q_0	q_1
q_1	q_2	q_0
q_2	q_3	q_0
q_3	q_3	q_3

d) (5 marks) Write an algorithm that uses the above table to decide if a string is acceptable by the DFA

i = 0

state = 0

while (input[i])

state = A[state, input[i++]];

if (state == q_3) display "accept"

else

display "reject"

Q 3) Consider the following NFA

a) (2 marks) Describe the language accepted by the above NFA using a regular expression.

$(ab)^+$

b) (4 marks) Convert the above NFA into a DFA.

Q 4) (6 marks) Write a CFG describing a language over the alphabet Σ={a,b}, containing all strings with

a) equal number of a's and b's.

$S \rightarrow aSbS \mid bSaS \mid \varepsilon$

b) the number of a's is double the number of b's.

$S \rightarrow aSbSaS \mid aSaSbS \mid bSaSaS$
$\mid \cancel{aSaSbS} \mid \cancel{aSbSaS} \mid \varepsilon$

28/32

Programming Language Compilation
Midterm-1
CSC 340
KSU (2nd term 2017-2018)

Student Name: [redacted] **Student Number:** [redacted]

Q1) a) Which programming language is better with respect to reliability Java or C++? Explain, why? (4 marks)

Java, because it check the ~~indexing~~ index of the array if it out of bound or not.

and C++ have Pointers and they are not reliable.

4

b) Which programming language is better with respect to the cost of execuation Java or C++? Give at least 2 reasons. (4 marks)

Answer: C++

Reasons:

1. C++ does not check for Proper indexing, it allows ~~to~~ the user to go out of bound in the array

2. ~~C++~~ the compiler of C++ translate to machine langage ~~imedi~~ directly, but in Java it transtute to intermadite langage then to machine langage.

4

Q2) a) Design a DFA that accepts a string if it belongs to the language $\{a^p b^q c^r \mid p, q, r \geq 0\}$.
(4 marks)

2

b) Consider the finite automaton in the following figure.

1. What is the set of reachable states for the input string 0011? (2 marks)

$\{q_0, q_1, q_2\}$

2

2. Is the string 0011 acceptable? Why? (2 marks)

2 NO, because the reachable state ~~are~~ with this input are $\{q_0, q_1, q_2\}$ and non of them ~~or~~ belongs to F

$\{q_0, q_1, q_2\} \notin F$ "set of final states"

d) Represent the above finite automaton using a table. (2 marks)

	0	1
$\rightarrow q_0$	q_0	$\{q_0, q_1\}$
q_1	q_2	q_2
q_2	q_3	$\{q_3, q_2\}$
$*q_3$	—	—

2

Q3) Consider the following NFA

a) Describe the language recognized by the NFA using a regular expression. (2 marks)

$a^* + (ab)^+$

2

b) Convert the following NFA into a DFA (4 marks)

4

ε-close$\{1\} = \{1, 2, 3\}$

NOTE

	a	b
$\rightarrow *\{1,2,3\}$	$\{2,4\}$	—
$*\{2,4\}$	$\{2\}$	$\{5,3\}$
$*\{2\}$	$\{2\}$	—
$*\{5,3\}$	$\{4\}$	—
$\{4\}$	—	$\{5,3\}$

c) Represent the DFA using a Table (1 mark)

	a	b
→*{1,2,3}	{2,4}	—
*{2,4}	{2}	{5,3}
*{2}	{2}	—
*{5,3}	{4}	—
{4}	—	{5,3}

1

d) Write an algorithm that takes a string over the alphabet and displays "accept" if the string is acceptable by the DFA, or "reject" if it is not. (3 marks)

```
inPut= input string
state =0   // start state
while (inPut){
state = table [state , inPut++]
}
if (state ∈ F)   // F is set of final states
    return ("accept")
else
    return ("reject")
```

2

e) Convert the regular expression $(0+1)^*$ over the alphabet {0,1} into an NFA (3 marks)

3

Midterm-1 (2nd term 2018-2019)
CSC 340
KSU

Student Name: [redacted]

Q1) a) Compare between C++ and Java with respect to reliability. (4 grades)

What language is better? Java

List features or constructs makes it so?

1) begouce chenk ~~the~~ index of the attay don't ~~dint~~ go out of bou attay.

2) hight execution cost.

b) Compare between C++ and Java with respect to the cost of execution. (4 grades)

What language is better? C++

List features or constructs makes it so?

1) dose not have chenk PotPet of index of atto Points ~~it~~ allow to the user go out of bound of a

for C++

2) tranlstion for mchien langug, Java ~~tran~~ tranlst ~~tranlaeit~~ tranlaeit langug then to mchien Langu

c) Discuss how the Von Neumann architecture influenced the design of modern programming languages. (4 grades)

1- get the intraction
2- Loop forever
3- Process
4- decodeing
5- end Loop

Q2)

A) Design a DFA that accepts a string containing letter and digits where the string consists of at most two characters; the first character must be a letter (A-Z), while the second may be a letter or digit (0-9). (4 grades)

RE = Letter

B) Write a regular expression that defines the token class identifier where an identifier can consist of at most two characters; the first character must be a letter (A-Z), while the second may be a letter or digit (0-9). (4 grades)

regular expressio is (Letter digit)

Q3) A) Convert the regular expression below into an NFA. (5 grades)

(a+b)*aab*

5

B) Consider the following NFA

1. Represent the NFA using a table. (3 grades)

	0	1	ε
q_0	—	q_1	q_1
q_1	$\{q_0, q_1\}$	—	q_2
q_2	—	—	—

2. What is the shortest string that would be rejected by the above NFA (2 grades)

zero is reject

.c Analysis

Write an algorithm that accepts a string if it is accepted by the above NFA. (6 grades)

```
i = 0;
state = 0;
while (input[i]) {
  state = table[state, input[i++]];
}
if (state ∈ F)
  return "accept";
else
  return "reject";
```

4. Convert the above NFA into a DFA. (4 grades)

ϵ-closure$(q_2) = \{q$

$\{q_0, q_1\}$	$\{q_0$
$\{q_1\}$	$\{q$

Programming Language Compilation
Midterm (1st term 2020-2021)
CSC 340
KSU

Student Number: **Student Name:**

Q1) a) Compare between Java and C++ with respect to reliability. (2 marks)

any 2 points sufficient

- Java performs array index checking; C++ does not
- Java uses references which are much cleaner than pointers as used in C++
- Java is a strongly typed language while C++ is weakly typed

⇒ Java is more reliable

b) Discuss how pointers in C++ affects

1) writability (1 marks)

Good for writability as they allow us to use vars, array, objects using their addresses in the memory

2) reliability (1 marks)

Bad in terms of reliability as mistakes can happen, especially if we use pointer arithmentics

c) Why were modular languages developed? (2 marks)

- To support huge applications
- divide and conquer approach for development
- data abstraction

Q2) Consider the following NFA

1. Describe the language accepted by this NFA using a regular expression. (1 marks)

$(a+b)^* abba$

2. Convert the above NFA into a DFA. (4 marks)

3. Represent the DFA using a table. (1 marks)

State	a	b
1	2	3
2	2	4
3	3	2
4	2	5
5	6	3
6	2	3

4. Write an algorithm that uses the above table to decide if a string is acceptable by the DFA or not. (2 marks)

```
i = 0
state = 0
while (input[i])
        state = T[state, input[i++]]

if state = 6
        then accept
else
        reject
```

Q3) a) Consider the alphabet the alphabet S= {a, b, c}.

A) Design a DFA that accepts a string in which every 'a' is followed by a 'b', but no 'b' is followed by a 'c'. For example, the DFA should accept "cabbab" but not "abc". (3 marks)

B) Write a regular expression that describes the language accepted by the above DFA. (2 marks)

$c^* b^* (ab)^*$

Q4) a) Consider the following CFG

E → E - int | E / int | int

a) Which operation has the highest precedence? (1 marks)

- the leftmost operation

b) What is the association of the – operation? (1 marks)

- Left to right

c) What is the association of the / operation? (1 marks)

- Left to right

d) Show that the string ***int-int/int*** belongs to the language of the above CFG (valid). (1 marks)

- either

E ⇒ E/int ⇒ E-int / int ⇒ int - int / int

or

e) Write a CFG that describes all mathematical expressions that can be formed using only * and +. Makes sure that * has higher precedence over + and the associativity of both operations is left to right. (3 marks)

E → E' + int | E'

E' → E' * int | E | int

Programming Language Compilation
Midterm-2
CSC 340
KSU (2nd term 2017-2018)

Student Name: **Student Number:**

Q1) A) Consider the following CFG

E→E+E|int

1) Show that the grammar is ambiguous. (2 marks)

the string int + int + int has more than one parse tree ⟹ the grammar is ambiguous

2) Rewrite the grammar to resolve ambiguity in such a way that + has left associativity. (3 marks)

E → E + int | int

B) Rewrite the following CFG so that we can write a recursive descent parser for.

E→E+E | E * E | id (2 marks)

* we need to eliminate left recursion

E → id X ε

X → + E X | * E X | ε

C) Rewrite the following CFG so that we can write an LL(1) for.

E→ id+E | id * E | (E) | id (2 marks)

* we need to left factor the grammar

E → id X | (E)

X → + E | * E | ε

D) Write a recursive descent parser in pseudo code (or c or Java code) for the grammar

E→ id+E | id * E | (E) | id (3 marks)

```
bool term(Token tok){ return *next++ == tok; }
E() { Token *save = next;
      return (next=save, E1() ||
              next=save, E2() ||
              next=save, E3() ||
              next=save, E4() )}
E1() { return term(id) && term(+) && E(); }
E2() - ---
E3() {}
E4() {}
```

Q2 A) Consider the following CFG

E→E+T|T
T→ T*F | F
F→ (E) | id

1. **Find the first and follow set of E** (2 marks)

First(E) = { (, id }

Follow(E) =

{ $, +,) }

2. **Find the first and follow set of F** (2 marks)

First(F) = { (, id }

Follow(F) =

{ $, +,), * }

3. **Find the first and follow set of ')'** (2 marks)

First(')') :

= { ')' }

Follow(')') : follow(F) = follow(T) ∪ follow(E)

{ $, +,), * }

B) Assume that a grammar contains the production E→T+F|TF. Assume also that first(T)={id, ε}, fist(F)= { * , ε }, follow(E)={id,$} and follow(T)={+, $}

1. Place the productions **E→T+F and E→TF in the right entries of the LL(1) table.** (3 marks)

	id	*	+	$
E	T+F / TF	TF	T+F	TF

2. **Is the table LL(1)? Why? Or Why not?** (2 marks)

Not LL(1) because entry [E, id] is multiply defined

Q3. **Assuming that there is a state in the DFA for recognizing the viable prefixes of a grammar contains the items.**

E→T.+F
T→F.

a) Under what condition would the SLR algorithm reduce? (2 marks)

if the next input token is in the follow

b) Under what condition would the SLR algorithm shift? (2 marks)

If the next input token is '+'

c) Assuming the follow(T)=follow(F)={+,*,$}, is the grammar SLR? Why or why not? (2 marks)

No because + ~~is in~~ '+' ∈ follow(T)

Q4) Draw the symbol table when line 14 in the following code is processed. (3 marks)

```
 0: int x = 137;
 1: int z = 42;
 2: int MyFunction(int x, int y) {
 3:   printf("%d,%d,%d\n", x, y, z);
 4:   {
 5:     int x, z;
 6:     z = y;
 7:     x = z;
 8:     {
 9:       int y = x;
10:       {
11:         printf("%d,%d,%d\n", x, y, z);
12:       }
13:       printf("%d,%d,%d\n", x, y, z);
14:     }
15:     printf("%d,%d,%d\n", x, y, z);
16:   }
17: }
```

x	0
z	1
x	2
y	2
x	5
z	5

Midterm-2 (1st term 2018-2019)
CSC 340
KSU

21/36

Student Name: [redacted]
Section:

Student Number: [redacted]
Seat Number:

Q1) Consider the following CFG (12 marks)

S→ int * E| int +E
E→ int

A) Write two left most derivation for the string int*int+int*int (2 marks)

First Derivation:

E → E*int | E+int
E → int

Second Derivation:

B) Since the string int*int+int*int has more than one leftmost derivations, can we conclude that the grammar is ambiguous. (2 marks)

~~the grammar~~
the string ~~can~~ can do only in one pars tree
so is unambiguous

C) Which operation has the highest precedence? Why? (2 marks)

is *, becuas, highest ~~Pre~~ Priority then +
Les

D) Is the associativity of the operation * right or left associativity? (2 marks)

right associativity

E) Rewrite the above grammar so that ~~you reverse~~ the * operation has higher precedence than the + operation and both operations have right associativity. (4 marks)

E → int E'

E' → * E | X | ε

X → + E | ε

Q2. Consider the following CFG (please notice that it is slightly different than the grammar in the previous question) (12 marks)

S→ S* E| E + S

E→ int | ε

a) Find the first and follow set for each terminal and non-terminal symbol. (4 marks)

Symbol	First Set	Follow Set
S	{int, ε}	{*, $}
E	{int, ε}	{+, *, $}
int	int	{+, *, $}
*	*	{int, *, $}
+	+	{int, *, $}

b) Draw the LL(1) parsing Table (3 marks)

	int	*	+	$
S	E+S	ε		ε
E	int	ε	ε	ε

c) Is the grammar LL(1) grammar? Why? or why not? (2 mark)

Yes, it's LL(1) grammar since you not have ~~contains~~ two Path in non ter

d) If your answer to the above question was no, then rewrite the grammar so that the grammar is LL(1). (3 marks)

eliminate left recursion.

Q3) Consider the CFG (16 marks)

S→ S* E| E + S
E→ int | ε

a) Design an NFA that recognizes the viable prefixes for the above grammar. (5 marks)

S' → S
S → S*E | E+S
E → int | ε

5

...an def f(x,y) = e the activation
follows:

b) Use the following LR(0) DFA to parse the string **(id, id + id)$** using the SLR parsing algorithm. Please use a suitable table to illustrate your answer. (5 marks)

5

.(id,id+id)$	1	shaft
(.id,id+id)$	3	shaft
(id.,id+id)$	2	reduce S→id
(S.,id+id)$	7	reduce L→S
(L.,id+id)$	5	shaft
(L,.id+id)$	8	chaft
(L,id.+id)$	2	reduce S→id
(L,S.+id)$	9	reduce L→L,S
(L.+id)$	9	shaft
(L+.id)$	3	~~shaft~~ reject
~~(L+id.)$~~	~~9~~	~~reject~~
~~(L+id)$~~	~~6~~	~~reduce S→(L+id)~~
~~S.$~~	~~4~~	~~S→(L+id)~~
~~S$.~~	~~final state~~	~~final state~~

c) Is the CFG represented using the above DFA an SLR grammar? Why? (2 marks)

easy to implement and ~~to~~ to ~~test the~~ input

2

d) Give two examples of problems that cannot be detected by both the lexical analyzer and the parser. (4 marks)

1.

2

2.