COMS W4115
Programming Languages and Translators
Lecture 1: January 23, 2013
Introduction to PLT

1. Teaching Staff

Instructor

Professor Alfred V. Aho
http://www.cs.columbia.edu/~aho
aho@cs.columbia.edu
513 Computer Science Building
Office hours: Mondays and Wednesdays 1:00-2:00pm
Course webpage: http://www.cs.columbia.edu/~aho/cs4115
Courseworks website: https://courseworks.columbia.edu
Piazza bulletin board: https://piazza.com/class#spring2013/comsw4115
Lectures on Mondays and Wednesdays, 2:40-3:55pm, 535 Mudd

TAs

Karan Bathla
kb2658@columbia.edu
Office hours: Mondays & Tuesdays 4:00-5:00
TA Room: 122 Mudd

Melanie Kambadur
melanie@cs.columbia.edu
Office hours: Thursdays 10:00-12:00
TA Room: 122 Mudd

Jared Pochtar
jrp2181@columbia.edu
Office hours: Fridays 3:00-5:00
TA Room: 122 Mudd

Maria Taku
mat2185@columbia.edu
Office hours: Tuesdays and Thursdays 12:30-2:30
TA Room: 122 Mudd

2. Course Objectives

- You will learn about the syntactic and semantic elements of modern programming languages.
- You will learn the important algorithms used by compilers to translate high-level source languages into machine and other target languages.
- You will learn about imperative, object-oriented, functional, logic, scripting languages, and parallel languages.
- A highlight of this course is a semester-long programming project in which you will work in a small team to create and implement an innovative little language of your own design.
- You will learn computational thinking and good software engineering practices.
- The concepts, techniques, and tools that you will learn in this course have broad application to many areas of computer science and software development outside of programming languages and compilers.

3. Course Syllabus

- Computational thinking
- Kinds of programming languages
- Principles of compilers
- Lexical analysis

- Syntax analysis
- Tools for constructing compilers
- Syntax-directed translation
- Semantic analysis
- Run-time organization
- Intermediate code generation
- Code generation
- Code optimization
- Parallel and concurrent programming languages

4. Textbooks and References

- The course text is
  - Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman
    Compilers: Principles, Techniques, and Tools, Second Edition
    Pearson Addison-Wesley, 2007

- Other good references are
  - Andrew W. Appel
    Modern Compiler Implementation in Java, second edition
    Cambridge University Press, 2002

  - Keith D. Cooper and Linda Torczon
    Engineering a Compiler, Second Edition
    Morgan Kaufmann, 2012

  - Steven S. Muchnick
    Advanced Compiler Design and Implementation
    Morgan Kaufmann, 1997

  - Michael L. Scott
    Programming Language Pragmatics, Third Edition
    Morgan Kaufman, 2009

  - Robert W. Sebesta
    Concepts of Programming Languages, Tenth Edition
    Pearson/Addison-Wesley, 2012

- Also see Professor Stephen Edwards' PLT website. Well worth a look!

5. Course Requirements, Grading, and Late Policy

- Homework (10% of final grade)
- Midterm (20% of final grade): Wednesday, March 13, 2013
- Final (30% of final grade): Monday, May 6, 2013
- Course project (40% of final grade): project has team and individual components
- All assignments can be handed in one week after they are due for 50% credit.

6. Project Requirements

- Form a team of five. Teams should be formed by Monday, February 4, 2013.
- Design a new innovative little language
- Build a compiler or interpreter for it.

- Project deliverables and due dates:

1. Feb 27: Language white paper. See Section 1.2 of http://www.oracle.com/technetwork/java/index-136113.html for a sample white paper on Java.
2. Mar 27: Language tutorial and reference manual.

- See Chapter 1 of K&R for a sample language tutorial.
- See Appendix A of K&R for a sample language reference manual.
- May 14-16: Final project report and demo to teaching staff.
- Start to form project teams of five right away. Elect a
- Project manager
- Language guru
- System architect
- System integrator
- Verification and validation person

7. Programming Languages

- A programming language is a notation for specifying computational tasks that a person can understand and a computer can execute.
- Every programming language has a syntax and a semantics.
  - The syntax specifies how a concept is expressed.
  - The syntax is often defined using a (context-free) grammar.

    statement -> `while (` *expression* `)` *statement*
  - The semantics specifies what the concept means or does.
  - Semantics can be specified operationally, axiomatically or denotationally.
- Ambiguity
  - "Time flies like an arrow."
- Domains of application
  - Scientific
    - Fortran
  - Business
    - COBOL
  - Artificial intelligence
    - LISP
  - Systems
    - C
  - Web
    - Java
  - General purpose
    - C++

8. Kinds of Languages

- Imperative
  - Specifies how a computation is to be done.
  - Examples: C, C++, C#, Fortran, Java
- Declarative
  - Specifies what computation is to be done.
  - Examples: Haskell, ML, Prolog
- von Neumann
  - One whose computational model is based on the von Neumann architecture.
  - Basic means of computation is through the modification of variables (computing via side effects).
  - Statements influence subsequent computations by changing the value of memory.
  - Examples: C, C++, C#, Fortran, Java
- Object-oriented
  - Program consists of interacting objects.
  - Each object has its own internal state and executable functions (methods) to manage that state.
  - Object-oriented programming is based on encapsulation, modularity, polymorphism, and inheritance.
  - Examples: C++, C#, Java, OCaml, Simula 67, Smalltalk
- Scripting
  - An interpreted language with high-level operators for "gluing together" computations.
  - Examples: AWK, Perl, PHP, Python, Ruby
- Functional

    - One whose computational model is based on the recursive definition of functions (lambda calculus).
    - Examples: Haskell, Lisp, ML.
- Parallel
    - One that allows a computation to run concurrently on multiple processors.
    - Examples
        - Libraries: POSIX threads, MPI
        - Languages: Ada, Cilk, OpenCL, Chapel, X10
        - Architecture: CUDA (parallel programming architecture for GPUs)
- Domain specific
    - Many areas have special-purpose languages to facilitate the creation of applications.
    - Examples
        - YACC for creating parsers
        - LEX for creating lexical analyzers
        - MATLAB for numerical computations
        - SQL for database applications
- Markup
    - Not programming languages in the sense of being Turing complete, but widely used for document preparation.
    - Examples: HTML, XHTML, XML

9. Influential Languages

- 1950s: assembler, Cobol, Fortran, Lisp
- 1960s: Algol60, Basic, Simula67
- 1970s: C, ML, scripting languages, application-specific languages
- See TIOBE Index for their list of this month's 100 most popular programming languages.
    - TIOBE top ten for January 2013: C, Java, Objective-C, C++, C#, PHP, Visual Basic, Python, Perl, Ruby

10. Language Design Issues

- Application domain
    - Exploit domain restrictions for expressiveness and performance.
- Computational model
    - Choose a model that encourages simplicity and ease of expression.
    - Incorporate a few primitives that can be elegantly combined to solve large classes of problems.
- Abstraction mechanisms
    - Abstractions should foster reuse and be suggestive of solutions.
- Type system
    - Type systems can help reliability and security of programs.
- Usability
    - Language design should promote readability, writability, and efficiency.

11. To Do

1. Start forming your project team immediately. Teams should be in place by Feb 4, 2013.
2. Use Courseworks discussion board (https://courseworks.columbia.edu) to publicize your interests.
3. Contact Maria Taku (mat2185@columbia.edu) for help forming or finding a team.
4. Give your language a name.

12. Reading Assignment

- ALSU: Chapters 1 and 2

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 2: January 28, 2013
Basic Elements of Languages and Compilers

Overview

1. Course project
2. What's in a language specification?
3. The C programming language
4. Fundamental elements of programming languages
5. Language processing tools

1. Course Project

- Project description
  - Form a team of five. Contact Maria Taku (mat2185@columbia.edu) for help finding or forming a team. Once your team is complete, let Maria know who is on your team.
  - Create an innovative little language of your own design.
  - Write a compiler or interpreter for your language.
  - Each team member must write at least 500 lines of code of the compiler or interpreter.
  - Project constitutes 40% of final grade.
- Project team: elect one person to serve each of the following functions.
  - Project manager
    - This person sets the project schedule, holds weekly meetings with the entire team, maintains the project log, and makes sure the project deliverables get done on time.
  - Language and tools guru
    - This person defines the baseline process to track language changes and maintain the intellectual integrity of the language.
    - This person teaches the team how to use various tools used to build the compiler.
  - System architect
    - This person defines the compiler architecture, modules, and interfaces.
  - System integrator
    - This person defines the system integration environment and makes sure the compiler components work together.
  - Tester and validator
    - This person defines the test suites and executes them to make sure the compiler meets the language specification.
- Project due dates and deliverables:
  - Feb. 27: Language white paper (written by entire team, 3-4 pages).
    - See http://www.oracle.com/technetwork/java/index-136113.html for a sample white paper on Java.
  - Mar. 27: Language tutorial (written by entire team, 15-20 pages).
    - Chapter 1 of Kernighan and Ritchie is a good model of a language tutorial.
    - Describe a few representative programs that illustrate the nature and scope of your language.
    - A "hello, world" program is de rigueur.
  - Mar. 27: Language reference manual (written by entire team, 20-25 pages).
    - Appendix A of Kernighan and Ritchie is a good model.
    - Give a complete description of the lexical and syntactic structure of your language.
    - Include a full grammar for your language.
  - May 14-16: Working compiler and demo.
  - May 14-16: Final project report due at project demo.

2. A Language Specification Defines

- the representation of programs
- the syntax and constraints of the language
- the semantic rules for interpreting programs
- the representation of input data to be processed by programs
- the representation of output data produced by programs
- other restrictions on programs (such as what makes a program portable across different implementations and platforms)

3. The C Programming Language

- C is a general-purpose procedural programming language that was designed in 1969-1972 at Bell Labs by Dennis Ritchie who was working on developing the Unix operating system with Ken Thompson. It is still one of the most widely used programming languages in the world.
- C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11.
- In 1978 Brian Kernighan and Dennis Ritchie published the book "The C Programming Language" which for many years served as the informal definition of C ("K&R C").

- Kernighan and Ritchie described C as "a general-purpose programming language which features economy of expression, modern control flow and data structures, and a rich set of operators."
- C has gone through a number of versions since K&R C:
  - ANSI C (1989) and ISO C (1990): these versions are identical and are commonly referred to as C89
  - C99: ISO/IEC9899:1999
  - C11, the current standard, adopted in 12/8/2011
  - Embedded C: in 2008 the C Standards Committee extended C to create programs to meet the stringent requirements of microcontroller systems

- A sample C program

```
(1)  /* this program computes the greatest common divisor
(2)     of two integers entered on the command line */

(3)  #include <stdio.h>
(4)  #include <stdlib.h>

(5)  int gcd(int m, int n)
(6)  {
(7)    int r;
(8)    while ((r = m % n) != 0) {
(9)      m = n;
(10)     n = r;
(11)   }
(12)   return n;
(13) }

(14) int main(int argc, char *argv[])
(15) {
(16)   int m, n;
(17)   m = atoi(argv[1]);
(18)   n = atoi(argv[2]);
(19)   printf("gcd of %d and %d is %d\n", m, n, gcd(m, n));
(20)   return 0;
(21) }
```

4. Fundamental Elements of Programming Languages

- Programming model
  - The programming model is the model of computation encapsulated into the programming language.
    - For example, C is an imperative language, designed around the von Neumann model of computation.
- Program structure
  - A program typically consists of one or more translation units stored in files.
    - In C, a translation unit is a sequence of function definitions and declarations.
- Character set and lexical conventions
  - Source and target character sets may be different.
    - The character set of C source programs in contained within seven-bit ASCII.
  - A token is a meaningful sequence of characters in a source program.
    - C has six classes of tokens: identifiers, keywords, constants, string literals, operators, and separators.
- Names, scopes, bindings, and lifetimes
  - Names (often called identifiers) have a specified lexical structure.
    - In C identifiers are sequences of letters (here, underscore is considered a letter) and digits. The first character of an identifier must be a letter. At least the first 31 characters in an identifier are significant.
  - The scope of a name is the region of the program in which it is known (visible).
  - A binding is an association between two things such as between a variable and its type or between a symbol and the operation it represents. The time at which this association is determined is called the binding time. Bindings can take place at various times ranging from language design time to run time.
  - The lifetime of a variable is the time during which the variable is bound to a specific memory location.
- Memory management
  - One important function of a programming language is to provide facilities for managing memory and the objects stored in memory.
    - C provides three ways to allocate memory for objects:
      - Static allocation where space for an object is provided by the compiler at run time.

        - Automatic allocation where temporary objects are stored by the compiler on the runtime stack. This space is automatically freed by the compiler after the block in which the object is declared is exited.
        - Dynamic allocation where blocks of memory of arbitrary size can be requested by a programmer using runtime library functions such as malloc from a region of memory called the heap. These blocks persist until freed by a call to a runtime library function such as free.
- Data types and operators
    - A data type defines a set of data values and the operations allowed on those values.
        - C has a small number of basic types, including `char, int, double, float, enum, void`.
        - C has a potentially infinite number of recursively defined derived types such as arrays of objects of some type, functions returning objects of some type, pointers to objects of some type, structures containing a sequence of objects of various types, and unions containing any one of several objects of various types.
        - C has a rich set of arithmetic, relational, logical, and assignment operators.
- Expressions and assignment statements
    - Expressions are the primary means for specifying computations in a programming language.
    - Assignment statements are basic constructs in imperative programming languages. Assignment statements allow the programmer to dynamically change the bindings of values to variables.
- Control flow
    - Flow of control refers to the sequence in which the operations specified in a program are executed at run time. There are flow-of-control issues at many levels such as flow of control within expressions, among statements, and among program units. Most programming languages have control statements and other control structures for controlling the flow of control within a program.
        - C has a variety of flow-of-control constructs such as blocks and control statements such as `if-else, switch, while, for, do-while, break, continue` and `goto`.
- Functions and process abstraction
    - Functions are perhaps the most important building blocks of programs. Functions are often called procedures, subroutines, or subprograms. Functions break large computing tasks into smaller ones and facilitate code reuse. Functions are such an important topic in programming languages that we will talk about them in much more detail later in this course.
- Data abstraction and object orientation
    - Data abstraction in the form of abstract data types was introduced into programming languages after process abstraction. The programming language Simula67 was instrumental in motivating constructs for supporting object-oriented programming in modern programming languages such as C++, C#, and Java. Object orientation is characterized by encapsulation, polymorphism, inheritance, and dynamic binding.
- Concurrency
    - Concurrent execution of programs has assumed much more importance with the widespread use of multi-core and many-core processors.
    - Concurrency in software execution can occur many levels of granularity: instruction, statement, subprogram, and program.
    - Concurrency can be achieved with libraries (like MPI for Fortran, pthreads for C) or with direct language support (as in Cilk, X10).
    - However, effective exploitation of concurrency is still an open research area in software.
- Exception and event handling
    - Many languages have facilities for reacting to run-time error conditions. C++ has the `try-catch` construct to catch exceptions raised by the `throw` statement.
    - Event handling is like exception handling in that an event handler is called by the occurence of an event. Implementing reactions to user interactions with GUI components is a common application of event handling.

5. Language Processing Tools

- Basic compiler
- Interpreter
- Bytecode interpreter
- Just-in-time compiler
- Linker and loader
- Preprocessor

6. Practice Problems

1. Describe the von Neumann model of computation (computer architecture).
2. Compare C and Java with regard to their (a) programming model and (b) primitive data types.

7. Reading Assignment

- ALSU: Chapters 1 and 2

8. Reference

- Brian Kernighan and Dennis Ritchie, *The C Programming Language*, 2nd edition, Prentice Hall, 1988. This is the classic reference on ANSI C (C89).

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 3: January 30, 2013
Structure of a Compiler

Overview

1. Language processing tools
2. Structure of a compiler
3. The lexical analyzer
4. Language theory background
5. Regular expressions
6. Tokens/patterns/lexemes/attributes

1. Language Processing Tools

- Compiler
- Interpreter
- C compiler
- Java compiler
- Just-in-time compiler

2. Structure of a Compiler

- Front end: analysis
- Back end: synthesis
- IR: Intermediate representation(s)
- Phases
  - lexical analyzer (scanner)
  - syntax analyzer (parser)
  - semantic analyzer
  - intermediate code generator
  - code optimizer
  - code generator
  - machine-specific code optimizer
- Symbol table
- Error handler
- Compiler component generators
  - lexical analyzer generators: lex, flex
  - syntax analyzer generator: yacc, bison
  - front-end generator: antlr

3. The Lexical Analyzer

- The first phase of the compiler is the lexical analyzer, often called a lexer or scanner.
- The lexer reads the stream of characters making up the source program and groups the characters into logically meaningful sequences called lexemes.
- Many lexers use a leftmost-longest rule. For example, a+++++b would be partitioned into the lexemes a `++` `++` `+` b, not a `++` `+` `++` b.
- For each lexeme the lexer sends to the parser a token of the form <token-name, attribute-value>.
- For a token such as an identifier, the lexer will make an entry into the symbol table in which it stores attributes such as the lexeme and type associated with the token.
- The lexer will also strip out whitespace (blanks, horizontal and vertical tabs, newlines, formfeeds, comments).
- Tokens in C
  - identifiers
  - keywords
  - constants
  - string literals
  - operators

    - separators
- Issues in the design of a lexical analyzer
    - efficiency: buffered reads
    - portability and character sets
    - need for lookahead
- Coping with lexical errors
    - types of lexical errors
        - insertion/deletion/replacement/transposition errors
    - edit distance
    - panic mode of error recovery

4. Language Theory Background

- Symbol (character, letter)
- Alphabet: a finite nonempty set of characters
    - Examples: {0, 1}, ASCII, Unicode
- String (sentence, word): a finite sequence of characters, possibly empty.
- Language: a (countable) set of strings, possibly empty.
- Operations on strings
    - concatenation
    - exponentiation
        - x^0 is the empty string ε.
        - $x^i = x^{i-1}x$, for $i > 0$
    - prefix, suffix, substring, subsequence
- Operations on languages
    - union
    - concatenation
    - exponentiation
        - L^0 is { ε }, even when L is the empty set.
        - $L^i = L^{i-1}L$, for $i > 0$
    - Kleene closure
        - $L^* = L^0 \cup L^1 \cup \ldots$
        - Note that L^* always contains the empty string.

5. Regular Expressions

- A regular expression is a notation for specifying a set of strings.
- Many of today's programming languages use regular expressions to match patterns in strings.
    - E.g., awk, flex, lex, java, javascript, perl, python
- Definition of a regular expression and the language it denotes
    - Basis
        - ε is a regular expression that denotes { ε }.
        - A single character *a* is a regular expression that denotes { *a* }.
    - Induction: suppose *r* and *s* are regular expressions that denote the languages L(*r*) and L(*s*).
        - (*r*)|(*s*) is a regular expression that denotes L(*r*) ∪ L(*s*).
        - (*r*)(*s*) is a regular expression that denotes L(*r*)L(*s*).
        - (*r*)* is a regular expression that denotes L(*r*)*.
        - (*r*) is a regular expression that denotes L(*r*).
    - We can drop redundant parenthesis by assuming
        - the Kleene star operator * has the highest precedence and is left associative
        - concatenation has the next highest precedence and is left associative
        - the union operator | has the lowest precedence and is left associative
    - E.g., under these rules r|s*t is interpreted as (r)|((s)*(t)).
    - Extensions of regular expressions
        - Positive closure: *r*+ = *rr**
        - Zero or one instance: *r*? = ε | *r*
        - Character classes:
            - [abc] = a | b | c
            - [0-9] = 0 | 1 | 2 | … | 9
- Today regular expressions come many different forms.

- The earliest and simplest are the Kleene regular expressions: See ALSU, Sect. 3.3.3.
- Awk and egrep extended grep's regular expressions with union and parentheses.
- POSIX has a standard for Unix regular expressions.
- Perl has an amazingly rich set of regular expression operators.
- Python uses pcre regular expressions.

- Lex regular expressions
  - The lexical analyzer generators flex and lex use extended regular expressions to specify lexeme patterns making up tokens: See ALSU, Fig. 3.8, p. 127.

6. Tokens/Patterns/Lexemes/Attributes

- a *token* is a pair consisting of a token name and an optional attribute value.
  - e.g., <id, ptr to symbol table>, <=>
- a *pattern* is a description of the form that the lexemes making up a token in a source program may have.
  - We will use regular expressions to denote patterns.
  - e.g., identifiers in C: `[_A-Za-z][_A-Za-z0-9]*`
- a lexeme is a sequence of characters that matches the pattern for a token, e.g.,
  - identifiers: `count, x1, i, position`
  - keywords: `if`
  - operators: `=, ==, !=, +=`
- an *attribute* of a token is usually a pointer to the symbol table entry that gives additional information about the token, such as its type, value, line number, etc.

7. Practice Problems

1. What language is denoted by the following regular expressions?
   a. `(a*b*)*`
   b. `a(a|b)*a`
   c. `(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*`
   d. `a(ba|a)*`
   e. `ab(a|b*c)*bb*a`
2. Construct Lex-style regular expressions for the following patterns.
   a. All lowercase English words with the five vowels in order.
   b. All lowercase English words with exactly one vowel.
   c. All lowercase English words beginning and ending with the substring "ad".
   d. All lowercase English words in which the letters are in strictly increasing alphabetic order.
   e. Strings of the form `abxba` where `x` is a string of a's, b's, and c's that does not contain `ba` as a substring.

8. Reading Assignment

- ALSU: Ch. 1, Sects. 3.1-3.3
- See The Lex & Yacc Page for lex, flex, yacc and bison tutorials and manuals.
- See ANTLR 3.x for an antlr video tutorial.

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 4: Regular Expressions and Lexical Analysis
February 4, 2013

Lecture Outline

1. Regular expressions
2. Lex regular expressions
3. Specifying a lexical analyzer with Lex
4. Example Lex programs
5. Creating a lexical processor with Lex

6. Lex history

1. Regular Expressions

- Regular expressions in various forms are used in many programming languages and software tools to specify patterns and match strings.
- Regular expressions are well suited for matching lexemes in programming languages.
- In formal language theory regular expressions use a finite alphabet of symbols and the operators union, concatenation, and Kleene closure. They define the regular languages.
- Unix programs like egrep, awk, and Lex extend this simple notation with additional operators and shorthands.
- The POSIX (Portable Operating System Inteface for Unix) standard defines two flavors of regular expressions for Unix systems: Basic Regular Expressions and Extended Regular Expressions.
- Perl has amazingly rich regular expressions which further extend the egrep, awk, and Lex regular expressions. Perl compatible regular expressions have been adopted by Java, JavaScript, PHP, Python, and Ruby.
- The back-referencing operator in Perl regular expressions allows nonregular languages to be recognized and makes the pattern-matching problem NP-complete.

2. Lex Regular Expressions

- The declarative language Lex has been widely used for creating many useful lexical analysis tools including lexers.
- The following symbols in Lex regular expressions have special meanings:

  `\ " . ^ $ [ ] * + ? { } | ( ) /`

  To turn off their special meaning, precede the symbol by `\`.
  - Thus, `\*` matches `*`.
  - `\\` matches `\`.
- Examples of Lex regular expressions and the strings they match.

1. `"a.*b"` matches the string `a.*b`.
2. `.` matches any character except a newline.
3. `^` matches the empty string at the beginning of a line.
4. `$` matches the empty string at the end of a line.
5. `[abc]` matches an `a`, or a `b`, or a `c`.
6. `[a-z]` matches any lowercase letter between `a` and `z`.
7. `[A-Za-z0-9]` matches any alphanumeric character.
8. `[^abc]` matches any character except an `a`, or a `b`, or a `c`.
9. `[^0-9]` matches any nonnumeric character.
10. `a*` matches a string of zero or more a's.
11. `a+` matches a string of one or more a's.
12. `a?` matches a string of zero or one a's.
13. `a{2,5}` matches any string consisting of two to five a's.
14. `(a)` matches an `a`.
15. `a/b` matches an `a` when followed by a `b`.
16. `\n` matches a newline.
17. `\t` matches a tab.

- Lex chooses the longest match if there is more than one match. E.g., `ab*` matches the prefix `abb` in `abbc`.

3. Specifying a Lexical Analyzer with Lex

- Lex is a special-purpose programming language for creating programs to process streams of input characters.
- Lex has been widely used for contructing lexical analyzers.
- A Lex program has the following form:

```
 declarations
%%
translation rules
%%
auxiliary functions
```

- The declarations section can contain declarations of variables, manifest constants, and regular definitions. The declarations section can be empty.
- The translation rules are each of the form

```
 pattern        {action}
```

- Each pattern is a regular expression which may use regular definitions defined in the declarations section.
- Each action is a fragment of C-code.

- The auxiliary functions section starting with the second %% is optional. Everything in this section is copied directly to the file `lex.yy.c` and can be used in the actions of the translation rules.

4. Example Lex programs

Example 1: Lex program to print all words in an input stream

- The following Lex program will print all alphabetic words in an input stream:

```
%%
[A-Za-z]+        { printf("%s\n", yytext); }
.|\n                { }
```

- The pattern part of the first translation rule says that if the current prefix of the unprocessed input stream consists of a sequence of one or more letters, then the longest such prefix is matched and assigned to the Lex string variable `yytext`. The action part of the first translation rule prints the prefix that was matched. If this rule fires, then the matching prefix is removed from the beginning of the unprocessed input stream.
- The dot in pattern part of the second translation rule matches any character except a newline at the beginning of the unprocessed input stream. The `\n` matches a newline at the beginning of the unprocessed input stream. If this rule fires, then the character of the beginning of the unprocessed input stream is removed. Since the action is empty, no output is generated.
- Lex repeated applies these two rules until the input stream is exhausted.

Example 2: Lex program to print number of words, numbers, and lines in a file

```
        int num_words = 0, num_numbers = 0, num_lines = 0;
word     [A-Za-z]+
number   [0-9]+
%%
{word}   {++num_words;}
{number} {++num_numbers;}
\n       {++num_lines; }
.        { }
%%
int main()
{
 yylex();
 printf("# of words = %d, # of numbers = %d, # of lines = %d\n",
        num_words, num_numbers, num_lines );
}
```

Example 3: Lex program for some typical programming language tokens

- See ALSU, Fig. 3.23, p. 143.

```
%{ /* definitions of manifest constants */
  LT, LE,
  IF, ELSE, ID, NUMBER, RELOP */

/* regular definitions */
delim         [ \t\n]
ws         {delim}+
letter          [A-Za-z]
digit    [0-9]
id         {letter}({letter}|{digit})*
number          {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws}         { }
if         {return(IF);}
else         {return(ELSE);}
{id}         {yylval = (int) installID(); return(ID);}
```

```
{number} {yylval = (int) installNum(); return(NUMBER);}
"<"         {yylval = LT; return(RELOP); }
"<="         {yylval = LE; return(RELOP); }
%%
int installID()
{
 /* function to install the lexeme, whose first character
    is pointed to by yytext, and whose length is yyleng,
    into the symbol table; returns pointer to symbol table
    entry */
}

int installNum() {
 /* analogous to installID */
}
```

5. Creating a Lexical Processor with Lex

- Put lex program into a file, say `file.l`.
- Compile the lex program with the command:

```
lex file.l
```

This command produces an output file `lex.yy.c`.

- Compile this output file with the C compiler and the lex library `-ll`:

```
gcc lex.yy.c -ll
```

The resulting `a.out` is the lexical processor.

6. Lex History

- The initial version of Lex was written by Michael Lesk at Bell Labs to run on Unix.
- The second version of Lex with more efficient regular expression pattern matching was written by Eric Schmidt at Bell Labs.
- Vern Paxson wrote the POSIX-compliant variant of Lex, called Flex, at Berkeley.
- All versions of Lex use variants of the regular-expression pattern-matching technology described in Chapter 3 of ALSU.
- Today, many versions of Lex use C, C++, C#, Java, and other languages to specify actions.

7. Practice Problems

1. Write a Lex program that copies a file, replacing each nonempty sequence of whitespace consisting of blanks, tabs, and newlines by a single blank.
2. Write a Lex program that converts a file of English text to "Pig Latin." Assume the file is a sequence of words separated by whitespace. If a word begins with a consonant, move the consonant to the end of the word and add "ay". (E.g., "pig" gets mapped into "igpay".) If a word begins with a vowel, just add "ay" to the end. (E.g., "art" gets mapped to "artay".)

8. Reading Assignment

- ALSU, Sects. 3.3, 3.5
- See The Lex & Yacc Page for Lex and Flex tutorials and manuals.

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Homework Assignment #1
Submit solutions electronically on
Courseworks/COMSW4115/Assignments
by 2:40pm, February 13, 2013

Instructions

- You may discuss the questions with others but your answers must be your own words and your own code.
- Problems 1-4 are each worth 25 points.
- Solutions to these problems will be posted on Courseworks on February 20, 2013.
- This assignment may submitted electronically on Courseworks by 2:40pm, February 20, 2013 for 50% credit.

Problems

1. Construct Lex-style regular expressions for the following patterns. For each pattern, create a Lex program to find the first longest lowercase word in the dictionary satisfying that pattern. As part of your answer list your program and the longest word found. State what dictionary you used (e.g., on Linux systems `/usr/dict/words`). You can use any variant of Lex such as Flex, JLex, Ocamllex, PLY, etc.
   a. All lowercase English words that can be made up using only the letters in your first name. Each letter in your name can be used zero or more times and in any order in a word.
   b. All lowercase English words that can be made up using exactly the letters in your first name. Each letter in your name must be used exactly the number of times it appears in your name but the letters can appear in any order in a word. If your name has more than five letters, use the prefix of length five.

2. Let *L* be the language consisting of all strings of *a*'s, *b*'s, and *c*'s such that each string is of the form *abxba* where *x* does not contain *ba* as a substring. These strings model comments in the programming language C.
   a. Write down a deterministic finite automaton (DFA) for *L*.
   b. Minimize the number states in your DFA. Briefly explain how you did the minimization.
   c. Explain what property of the scanned input each state of your minimized DFA recognizes.
   d. Write down a regular expression for *L*.

3. Let *L* be the language consisting of just balanced parentheses, i.e., {ε, `( )`, `( ( ) )`, `( ) ( )`, `( ( ( ) ) )`, `( ( ) ( ) )`, ...}
   a. Write down a recursive definition for *L*.
   b. Use the pumping lemma for regular languages to show that *L* cannot be specified by a regular expression.

4. Let *R* be a regular expression of length *m* and let *w* be an input string of length *n*. Briefly discuss in terms of *m* and *n* the time-space complexity of the McNaughton-Yamada-Thompson algorithm to determine whether *w* is in *L*(*R*).

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 5: Implementing a Lexical Analyzer
February 6, 2013

Outline

1. Finite automata
2. Converting an NFA to a DFA
3. Equivalence of regular expressions and finite automata
4. Simulating an NFA
5. The pumping lemma for regular languages
6. Closure and decision properties of regular languages

1. Finite Automata

- Variants of finite automata are commonly used to match regular expression patterns.
- A nondeterministic finite automaton (NFA) consists of
  - A finite set of states *S*.
  - An input alphabet consisting of a finite set of symbols Σ.
  - A transition function δ that maps $S \times (\Sigma \cup \{\varepsilon\})$ to subsets of *S*. This transition function can be represented by a transition graph in which the nodes are labeled by states and there is a directed edge labeled *a* from node *w* to node *v* if δ(*w*, *a*) contains *v*.
  - An initial state s_0 in *S*.

- F, a subset of S, called the final (or accepting) states.

- An NFA accepts an input string x iff there is a path in the transition graph from the initial state to a final state that spells out x.
- The language defined by an NFA is the set of strings accepted by the NFA.
- A deterministic finite automaton (DFA) is an NFA in which

1. There are no ε moves, and
2. For each state s and input symbol a there is exactly one transition out of s labeled a.

2. Converting an NFA to a DFA

- Every NFA can be converted to an equivalent DFA using the subset construction (Algorithm 3.20, ALSU, pp. 153-154).
- Every DFA can be converted into an equivalent minimum-state DFA Using Algorithm 3.39, ALSU, pp. 181-183. All equivalent minimum-state DFAs are isomorphic up to state renaming.

3. Equivalence of Regular Expressions and Finite Automata

- Regular expressions and finite automata define the same class of languages, namely the regular sets.
- Every regular expression can be converted into an equivalent NFA using the McNaughton-Yamada-Thompson algorithm (Algorithm 3.23, ALSU, pp. 159-161).
- Every finite automaton can be converted into a regular expression using Kleene's algorithm.

4. Simulating an NFA

- Two-stack simulation of an NFA: Algorithm 3.22, ALSU, pp. 156-159.

5. The Pumping Lemma for Regular Languages

- The pumping lemma allows us to prove certain languages, like $\{ a^n b^n \mid n \geq 0 \}$, are not regular.

  The pumping lemma. If L is a regular language, then there exists a constant n associated with L such that for every string w in L where $|w| \geq n$, we can partition w into three strings xyz (i.e., $w = xyz$) such that
  - y is not the empty string,
  - the length of xy is less than or equal to n, and
  - for all $k \geq 0$, the string $xy^k z$ is in L.

6. Closure and Decision Properties of Regular Languages

- The regular languages are closed under the following operations:
  - union
  - intersection
  - complement
  - reversal
  - Kleene star
  - homomorphism
  - inverse homomorphism
- Decision properties
  - Given a regular expression r and a string w, it is decidable whether r matches w.
  - Give a finite automaton A, it is decidable whether L(A) is empty.
  - Given two finite automata A and B, it is decidable whether L(A) = L(B).

7. Practice Problems

1. Write down deterministic finite automata for the following regular expressions:
   a. `(a*b*)*`
   b. `(aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*`
   c. `a(ba|a)*`
   d. `ab(a|b*c)*bb*a`
2. Construct a deterministic finite automaton that will recognize all strings of 0's and 1's representing integers that are divisible by 3. Assume the empty string represents 0.
3. Use the McNaughton-Yamada-Thompson algorithm to convert the regular expression `a(a|b)*a` into a nondeterministic finite automaton.
4. Convert the NFA of (3) into a DFA.
5. Minimize the number of states in the DFA of (4).

8. Reading Assignment

- ALSU Chapter 3, all sections except 3.9.
- Russ Cox's article Regular Expression Matching Can Be Simple and Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...) has a good historical account on the evolution of regular expression matching programs.

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 6: Context-Free Grammars
February 11, 2013

Lecture Outline

- Context-free grammars
- Derivations and parse trees
- Ambiguity
- Examples of context-free grammars
- Yacc: a language for specifying syntax-directed translators

1. Context-Free Grammars (CFG's)

- CFG's are very useful for representing the syntactic structure of programming languages.
- A CFG is sometimes called Backus-Naur Form (BNF).
- A context-free grammar consists of

1. A finite set of terminal symbols,
2. A finite nonempty set of nonterminal symbols,
3. One distinguished nonterminal called the start symbol, and
4. A finite set of rewrite rules, called productions, each of the form $A \rightarrow \alpha$ where A is a nonterminal and α is a string (possibly empty) of terminals and nonterminals.

- Consider the context-free grammar G with the productions

```
E → E + T | T
T → T * F | F
F → ( E ) | id
```

- The terminal symbols are the alphabet from which strings are formed. In this grammar the set of terminal symbols is { id, +, *, (,) }. The terminal symbols are the token names.
- The nonterminal symbols are syntactic variables that denote sets of strings of terminal symbols. In this grammar the set of nonterminal symbols is { E, T, F}.
- The start symbol is E.

2. Derivations and Parse Trees

- *L*(G), the language generated by a grammar G, consists of all strings of terminal symbols that can be derived from the start symbol of G.
- A leftmost derivation expands the leftmost nonterminal in each sentential form:

```
E ⇒ E + T
  ⇒ T + T
  ⇒ F + T
  ⇒ id + T
  ⇒ id + T * F
  ⇒ id + F * F
  ⇒ id + id * F
  ⇒ id + id * id
```

- A rightmost derivation expands the rightmost nonterminal in each sentential form:

```
E ⇒ E + T
  ⇒ E + T * F
  ⇒ E + T * id
  ⇒ E + F * id
  ⇒ E + id * id
  ⇒ T + id * id
  ⇒ F + id * id
  ⇒ id + id * id
```

- Note that these two derivations have the same parse tree.

3. Ambiguity

- Consider the context-free grammar G with the productions

```
E → E + E | E * E | ( E ) | id
```

This grammar has the following leftmost derivation for `id + id * id`

```
E ⇒ E + E
  ⇒ id + E
  ⇒ id + E * E
  ⇒ id + id * E
  ⇒ id + id * id
```

This grammar also has the following leftmost derivation for `id + id * id`

```
E ⇒ E * E
  ⇒ E + E * E
  ⇒ id +  E * E
  ⇒ id + id * E
  ⇒ id + id * id
```

- These derivations have different parse trees.
- A grammar is *ambiguous* if there is a sentence with two or more parse trees.
- The problem is that the grammar above does not specify
  - the precedence of the + and * operators, or
  - the associativity of the + and * operators
- However, the grammar in section (3) generates the same language and is unambiguous because it makes * of higher precedence than +, and makes both operators left associative.
- A context-free language is *inherently ambiguous* if it cannot be generated by any unambiguous context-free grammar.
- The context-free language $\{ a^m b^m a^n b^n \mid m > 0 \text{ and } n > 0 \} \cup \{ a^m b^n a^n b^m \mid m > 0 \text{ and } n > 0 \}$ is inherently ambiguous.
- Most (all?) natural languages are inherently ambiguous but no programming languages are inherently ambiguous.
- Unfortunately, there is no algorithm to determine whether a CFG is ambiguous; that is, the problem of determining whether a CFG is ambiguous is undecidable.
- We can, however, give some practically useful sufficient conditions to guarantee that a CFG is unambiguous.

4. Examples of Context-Free Grammars

- Nonempty palindromes of a's and b's. (A palindrome is a string that reads the same forwards as backwards; e.g., `abba`.)
  CFG: S → `a S b | b S a | a a | b b | a | b`
  Note that the language generated by this grammar is not regular. Can you prove this using the pumping lemma for regular languages?

- Strings with an equal number of a's and b's:
  CFG: S → `a S a | b S b | S S |` ε
  Note that this grammar is ambiguous. Can you find an equivalent unambiguous grammar?

- If- and if-else statements:

```
stmt → if ( expr ) stmt else stmt
     | if (expr) stmt
     | other
```

Note that this grammar is ambiguous.

- Some typical programming language constructs:

```
stmt → expr ;
     | if (expr) stmt
     | for ( optexpr; optexpr; optexpr;) stmt
     | other
optexpr → ε
     | expr
```

5. Yacc: a Language for Specifying Syntax-Directed Translators

- Yacc is popular language, created by Steve Johnson of Bell Labs, for implementing syntax-directed translators.
- Bison is a gnu version of Yacc, upwards compatible with the original Yacc, written by Charles Donnelly and Richard Stallman. Many other versions of Yacc are also available.
- The original Yacc used C for semantic actions. Yacc has been rewritten for many other languages including Java, ML, OCaml, and Python.
- Yacc specifications
  - A Yacc program has three parts:

```
 declarations
%%
translation rules
%%
supporting C-routines
```

The declarations part may be empty and the last part (%% followed by the supporting C-routines) may be omitted.

- Here is a Yacc program for a desk calculator that adds and multiplies numbers. (See ALSU, p. 292, Fig. 4.59 for a more advanced desk calculator.)

```
%{
#include <ctype.h>

#include <stdio.h>
#define YYSTYPE double
%}

%token NUMBER
%left '+'
%left '*'

%%

lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;

expr  : expr '+' expr      { $$ = $1 + $3; }
      | expr '*' expr      { $$ = $1 * $3; }
      | '(' expr ')'       { $$ = $2; }
      | NUMBER
      ;
```

```
%%
/* the lexical analyzer; returns <token-name, yylval> */
int yylex() {
  int c;
  while ((c = getchar()) == ' ');
  if ((c == '.') || (isdigit(c))) {
    ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
```

- The declarations

  `%left '+'`

  `%left '*'`

  make the operator + left associative and of lower precedence than the left-associative operator `*`.
- On Linux, we can make a desk calculator from this Yacc program as follows:

1. Put the yacc program in a file, say `desk.y`.
2. Invoke `yacc desk.y` to create the yacc output file `y.tab.c`.
3. Compile this output file with a C compiler by typing `gcc y.tab.c -ly` to get `a.out`. (The library -ly contains the Yacc parsing program.)
4. `a.out` is the desk calculator. Try it!

6. Practice Problems

1. Let G be the grammar S → a S b S | b S a S | ε.
   a. What language is generated by this grammar?
   b. Draw all parse trees for the sentence `abab`.
   c. Is this grammar ambiguous?
2. Let G be the grammar S → a S b | ε. Prove that $L(G) = \{ a^n b^n \mid n \geq 0 \}$.
3. Consider a sentence of the form `id + id + ... + id` where there are *n* plus signs. Let G be the grammar in section (3) above. How many parse trees are there in G for this sentence when *n* equals
   a. 1
   b. 2
   c. 3
   d. 4
   e. *m*?
4. Write down a CFG for regular expressions over the alphabet {a, b}. Show a parse tree for the regular expression a | b*a.

7. Reading

- ALSU Sects. 4.1-4.2, 4.9
- A nice Lex & Yacc tutorial

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 7: Parsing Context-Free Grammars
February 13, 2013

Outline

1. Yacc: a language for specifying syntax-directed translators

2. The pumping lemma for context-free languages
3. The parsing problem for context-free grammars
4. Top-down parsing
5. Transformations on grammars

1. Yacc: a Language for Specifying Syntax-Directed Translators

- Yacc is popular language, first implemented by Steve Johnson of Bell Labs, for implementing syntax-directed translators.
- Bison is a gnu version of Yacc, upward compatible with the original Yacc, written by Charles Donnelly and Richard Stallman. Many other versions of Yacc are also available.
- The original Yacc used C for semantic actions. Yacc has been rewritten for many other languages including Java, ML, OCaml, and Python.
- Yacc specifications
  - A Yacc program has three parts:

```
 declarations
%%
translation rules
%%
supporting C-routines
```

The declarations part may be empty and the last part (%% followed by the supporting C-routines) may be omitted.

- Here is a Yacc program for a desk calculator that adds and multiplies numbers. (From ALSU, p. 292, Fig. 4.59, a more advanced desk calculator.)

```
%{
#include <ctype.h>

#include <stdio.h>
#define YYSTYPE double
%}

%token NUMBER
%left '+'
%left '*'

%%

lines : lines expr '\n'   { printf("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;

expr  : expr '+' expr      { $$ = $1 + $3; }
      | expr '*' expr      { $$ = $1 * $3; }
      | '(' expr ')'       { $$ = $2; }
      | NUMBER
      ;

%%
/* the lexical analyzer; returns <token-name, yylval> */
int yylex() {
  int c;
  while ((c = getchar()) == ' ');
  if ((c == '.') || (isdigit(c))) {
    ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
```

- On Linux, we can make a desk calculator from this Yacc program as follows:

1. Put the yacc program in a file, say `desk.y`.
2. Invoke `yacc desk.y` to create the yacc output file `y.tab.c`.
3. Compile this output file with a C compiler by typing `gcc y.tab.c -ly` to get `a.out`. (The library -ly contains the Yacc parsing program.)
4. `a.out` is the desk calculator. Try it!

2. The Pumping Lemma for Context-Free Languages

- The pumping lemma for context-free languages can be used to show certain languages are not context free.

  The pumping lemma: If L is a context-free language, then there exists a constant *n* such that if z is any string in L of length *n* or more, then z can be written as uvwxy subject to the following conditions:

  1. The length of vwx is less than or equal to *n*.
  2. The length of vx is one or more. (That is, not both of v and x can be empty.)
  3. For all $i \geq 0$, uv^iwx^iy is in L.
- A typical proof using the pumping lemma to show a language L is not context free proceeds by assuming L is context free, and then finding a long string in L which, when pumped, yields a string not in L, thereby deriving a contradiction.
- Examples of non-context-free languages:
  - $\{a^nb^nc^n \mid n \geq 0\}$
  - {ww | w is in (a|b)* }
  - $\{a^mb^na^mb^n \mid n \geq 0\}$

3. The Parsing Problem for Context-Free Grammars

- The parsing problem for context-free grammars is given a CFG G and an input string w to construct all parse trees for w according to G, if w is in L(G).
- The Cocke-Younger-Kasami algorithm is a dynamic programming algorithm that given a Chomsky Normal Form grammar G and an input string w will create in $O(|w|^3)$ time a table from which all parse trees for w according to G can be constructed.
- For compiler applications two styles of parsing algorithms are common: top-down parsing and bottom-up parsing.

4. Top-Down Parsing

- Top-down parsing consists of trying to construct a parse tree for an input string starting from the root and creating the nodes of the parse tree in preorder.
- Equivalently, top-down parsing consists of trying to find a leftmost derivation for the input string.
- Consider grammar *G*:

```
S → + S S | * S S | a
```

- Leftmost derivation for + a * a a:

```
S ⇒ + S S
  ⇒ + a S
  ⇒ + a * S S
  ⇒ + a * a S
  ⇒ + a * a a
```

- Recursive-descent parsing
  - Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input string.
  - One procedure is associated with each nonterminal of the grammar. See Fig. 4.13, p. 219.
  - The sequence of successful procedure calls defines the parse tree.
- Nonrecursive predictive parsing
  - A nonrecursive predictive parser uses an explicit stack.
  - See Fig. 4.19, p. 227, for a model of table-driven predictive parser.
  - Parsing table for *G*:

```
                      Input Symbol
  Nonterminal    a        +          *        $
```

```
S        S → a    S → +SS    S → *SS
```

- Moves made by this predictive parser on input `+a*aa`. (The top of the stack is to the left.)

```
Stack        Input   Output
 S$         +a*aa$
 +SS$       +a*aa$   S → +SS
 SS$         a*aa$
 aS$         a*aa$   S → a
 S$           *aa$
 *SS$         *aa$   S → *SS
 SS$           aa$
 aS$           aa$   S → a
 S$             a$
 a$             a$   S → a
 $               $
```

- Note that these moves trace out a leftmost derivation for the input.

5. Transformations on Grammars

- Two common language-preserving transformations are often applied to grammars to try to make them parsable by top-down methods. These are eliminating left recursion and left factoring.
- Eliminating left recursion:
  - Replace

```
expr →  expr + term
     |  term
```

by

```
expr  →  term expr'

expr' →  + term expr'
      |  ε
```

- Left factoring:
  - Replace

```
stmt → if ( expr ) stmt else stmt
     | if (expr) stmt
     | other
```

by

```
stmt  → if ( expr ) stmt stmt'
      | other

stmt' → else stmt
      | ε
```

6. Practice Problems

1. Write down a CFG for regular expressions over the alphabet {a, b}. Show a parse tree for the regular expression `a | b*a`.
2. Using the nonterminals `stmt` and `expr`, design context-free grammar productions to model
   a. C while-statements
   b. C for-statements
   c. C do-while statements

3. Consider grammar *G*:

   S → S S + | S S * | a

   a. What language does this grammar generate?
   b. Eliminate the left recursion from this grammar.
4. Use the pumping lemma to show that $\{a^n b^n c^n \mid n \geq 0\}$ is not context free.

7. Reading

- ALSU, Sections 4.3, 4.4, 4.9.
- See The Lex & Yacc Page for lex and yacc tutorials and manuals.
- Another nice Lex & Yacc tutorial

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 9: Predictive Top-Down Parsers
February 20, 2013

Lecture Outline

1. Review
2. FIRST
3. FOLLOW
4. How to construct a predictive parsing table
5. LL(1) grammars
6. Transformations on grammars

1. Review

- Top-down parsing consists of constructing or tracing a parse tree for an input string starting from the root and creating the nodes of the parse tree in preorder.
- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input string with a procedure associated with each nonterminal of the grammar. See Fig. 4.13, p. 219.
- A nonrecursive predictive parser uses an explicit stack and a parsing table to do deterministic top-down parsing.
- In this class we will develop an algorithm to construct a predictive parsing table for a large class of useful grammars called LL(1) grammars.
- For this algorithm we need two functions on grammars, FIRST and FOLLOW.

2. FIRST

- FIRST(α) is the set of terminal symbols that begin the strings derivable from a string of terminal and nonterminal symbols α in a grammar.
  If α can derive ε, then ε is also in FIRST(α).
- Algorithm to compute FIRST(*X*):

1. If *X* is a terminal, then FIRST(*X*) = { *X* }.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(*X*).
3. If $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for $k \geq 1$, and
   for some $i \leq k$, $Y_1 Y_2 \ldots Y_{i-1}$ derives the empty string, and a is in FIRST(Y_i), then add a to FIRST(*X*).
   If $Y_1 Y_2 \ldots Y_k$ derives the empty string, then add ε to FIRST(*X*).

- **Example.** Consider the grammar *G*:

  S → (S) S | ε

  For *G*, FIRST(S) = {(, ε}.

3. FOLLOW

- FOLLOW(*A*) is the set of terminals that can appear immediately to the right of *A* in some sentential form in a grammar.
  Let us assume the string to be parsed is terminated by an end-of-string endmarker $. Then if *A* can be the rightmost symbol in some sentential form, the right endmarker $ is also in FOLLOW(*A*).
- Algorithm to compute FOLLOW(*A*) for all nonterminals *A* of a grammar:

1. Place $ in FOLLOW(*S*) where *S* is the start symbol of the grammar.
2. If $A \to \alpha B \beta$ is a production, then add every terminal symbol a in FIRST(β) to FOLLOW(*B*).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ε,
   then add every symbol in FOLLOW(*A*) to FOLLOW(*B*).

- **Example.** For *G* above, FOLLOW(S) = {), $}.

4. How to Construct a Predictive Parsing Table

- Input: Grammar *G*.
- Output: Predictive parsing table *M*.
- Method:

```
for (each production A → α in G) {
   for (each terminal a in FIRST(α))
      add A → α to M[A, a];
   if (ε is in FIRST(α))
      for (each symbol b in FOLLOW(A))
         add A → α to M[A, b];
}
make each undefined entry of M be error;
```

- **Example 1.** Predictive parsing table for the grammar:

  $S \to +SS \mid *SS \mid a;$

FIRST(S) = {+, *, a}
FOLLOW(S) = {+, *, a, $}

	Input Symbol			
Nonterminal	a	+	*	$
S	S → a	S → +SS	S → *SS	**error**

- **Example 2.** Predictive parsing table for the grammar:

  $S \to (S) S \mid \varepsilon$

FIRST(S) = {(, ε}
FOLLOW(S) = {), $}

	Input Symbol		
Nonterminal	(	)	$
S	S → (S)S	S → ε	S → ε

- **Example 3.** Predictive parsing table for the grammar:

  $S \to S (S) \mid \varepsilon$

FIRST(S) = {(, ε}
FOLLOW(S) = {(,), $}

	Input Symbol		
Nonterminal	(	)	$
S	S → S(S) S → ε	S → ε	S → ε

5. LL(1) Grammars

- A grammar is LL(1) iff whenever A $\rightarrow$ α | β are two distinct productions, the following three conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If β derives the empty string, then α does not derive any string beginning with a terminal in FOLLOW(A).
   Likewise, if α derives the empty string, then β does not derive any string beginning with a terminal in FOLLOW(A).

- We can use the algorithm above to construct a predictive parsing table with uniquely defined entries for any LL(1) grammar.
- The first "L" in LL(1) means scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one symbol of lookahead to make each parsing action decision.

6. Transformations on Grammars

- Two common language-preserving transformations are often applied to grammars to try to make them parsable by top-down methods. These are eliminating left recursion and left factoring.
- Eliminating left recursion:
  - Replace

```
expr →  expr + term
     |  term
```

by

```
expr  →  term expr'

expr' →  + term expr'
      |  ε
```

- Left factoring:
  - Replace

```
stmt → if ( expr ) stmt else stmt
     | if (expr) stmt
     | other
```

by

```
stmt  → if ( expr ) stmt stmt'
      | other

stmt' → else stmt
      | ε
```

7. Practice Problems

Consider the following grammar G for Boolean expressions:

```
B → B or T | T
T → T and F | F
F → not B | ( B ) | true | false
```

- What precedence and associativity does this grammar give to the operators `and, or, not`?
- Compute FIRST and FOLLOW for each nonterminal in G.
- Transform G into an equivalent LL(1) grammar G'.
- Construct a predictive parsing table for G'.
- Show how your predictive parser processes the input string

```
true and not false or true
```

Draw the parse tree traced out by your parser.

8. Reading

- ALSU, Section 4.4.

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 10: Bottom-Up Parsing
February 25, 2013

Lecture Outline

1. Bottom-up parsing
2. LR(1) parsing
3. Constructing a simple LR(1) parser
4. DFA for viable prefixes

1. Bottom-up Parsing

- Bottom-up parsing can be viewed as trying to find a rightmost derivation in reverse for an input string.
- A *handle* is a rightmost substring in right-sentential form that matches the body of a production and whose reduction by that production represents one step in the reverse of the rightmost derivation.
  - Consider the grammar *G*:

```
(1) S → S ( S )
(2) S → ε
```

- The handles in a rightmost derivation for the input string `( ) ( )`:

```
S ⇒ S ( S )      // handle is S ( S )
  ⇒ S ( )        // handle is the empty string between ( )
  ⇒ S ( S ) ( )  // handle is S ( S )
  ⇒ S ( ) ( )    // handle is the empty string between first ( )
  ⇒ ( ) ( )      // handle is the empty string prefix
```

- *Shift-reduce parsing* is a form of bottom-up parsing in which we shift terminal symbols of the string to be parsed onto a stack until a handle appears on top of the stack. We then replace the handle by the nonterminal symbol on the left-hand side of the associated production (this is a "reduce" action). We keep repeating this process until we have reduced the input string to the start symbol of the grammar. This process simulates the reverse of a rightmost derivation for the input string. Thus, we can think of shift-reduce parsing as "handle pruning."

2. LR(1) Parsing

- Model of an LR(1) parser (Fig. 4.35).
- "L" means left-to-right scanning of the input, the "R" means constructing a rightmost derivation in reverse, and the "1" means one symbol of lookahead in making parsing decisions.
- LR parsing table for *G*:

State	Action			Goto
	(	)	$	S
0	r2	r2	r2	1
1	s2		acc	
2	r2	r2	r2	3
3	s2	s4		
4	r1	r1	r1	

r2 means reduce the handle on top of the stack by production (2) S → ε.

s2 means shift the input symbol on the stack and then push state 2 on top of the stack.

acc means accept and stop parsing.

Goto[0,S] = 1 means push state 1 on top of the stack after reducing a handle to the nonterminal S in state 0.

A blank entry means report a syntax error.

- Moves made by an LR(1) parser on input ( ) ( ) [Alg. 4.44].

Stack	Input	Action
0	()()$	reduce by (2) S → ε; push state 1 on stack
0S1	()()$	shift (on stack; push state 2 on stack
0S1(2	)()$	reduce by (2) S → ε and push state 3
0S1(2S3	)()$	shift (and push state 2
0S1(2S3)4	()$	reduce by (1) S → S(S) and push state 1
0S1	()$	shift (and push state 2
0S1(2	)$	reduce by (2) S → ε and push state 3
0S1(2S3	)$	shift) and push state 4
0S1(2S3)4	$	reduce by (1) S → S(S) and push state 1
0S1	$	accept

- Note that an LR parser is a shift-reduce parser that traces out a rightmost derivation in reverse.

3. Constructing a Simple LR(1) Parsing Table for a Grammar

- An *LR*(0) *item* of a grammar is a production of the grammar with a dot at some position of the right side. E.g., S → ·S(S), S → S·(S), or S → S(S)·.
- We will use two functions to construct the sets of items for a grammar:
  - *closure*(*I*), where *I* is a set of items, is the set of items constructed by the following two rules:

1. Initially, put every item in *I* into *closure*(*I*).
2. If $A \to \alpha \cdot B\beta$ is in *closure*(*I*) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to *closure*(*I*) if it is not already there. Keep repeating this step until no more new items can be added to *I*.

- *goto*(*I*, *X*), where *I* is a set of items and *X* is a grammar symbol, is the closure of the set of all items $A \to \alpha X \cdot \beta$ where $A \to \alpha \cdot X\beta$ is in *I*.
- An *augmented* grammar *G′* is one to which we have added a new starting production *S′* → *S* where *S* is the start symbol of the given grammar *G*. Reducing by the new starting production signals acceptance of the input string being parsed. We will always augment a grammar when we construct an SLR parsing table for it.
- The sets-of-items construction
  - Input: An augmented grammar *G′*.
  - Output: *C*, the canonical collection of sets of LR(0) items for *G′*.
  - Method:

```
I0 = closure({[S' → ·S]});
C = {I0};
repeat
  for each set of items I in C and grammar symbol X such that
    goto(I,X) is not empty and not in C do
      add goto(I,X) to C;
until no more sets of items can be added to C;
```

- Example: Given the augmented grammar *G′*

```
S' → S
S  → S(S)
S  → ε
```

C, the canonical collection of sets of LR(0) items for *G′*, is

```
I0: S' → ·S
```

```
    S → ·S(S)
    S → ·

I1: S' → S·
    S → S·(S)

I2: S → S(·S)
    S → ·S(S)
    S → ·

I3: S → S(S·)
    S → S·(S)

I4: S' → S(S)·
```

- Algorithm to construct the SLR(1) parsing table from C, the canonical collection of sets of LR(0) items for an augmented grammar *G'*
  - Input: C = {I_0, I_1, ... , I_n}.
  - Output: The SLR parsing table functions action and goto.
  - Method:
    - State i and its action and goto functions are constructed from I_i as follows:
      - If item [A→ α·aβ] is in I_i and goto(I_i, a) = I_j, then add "shift j" to action[i, a]. Here a is a terminal.
      - If item [A → α·] is in I_i, then add "reduce A → α" to action[i, a] for all a in FOLLOW(A). Here A cannot be S'.
      - If item [S' → S·] is in I_i, then add "accept" to action[i, $].
    - If goto(I_i, A) = I_j, then in the parsing table set goto[i, A] = j.
    - The initial state of the parser is constructed from the set of items containing [S' → ·S].
  - Notes:
    - If each parsing table entry has at most one action, then the grammar is said to be *SLR(1)*. If any entry has more than one action, then the algorithm fails to produce a parser.
    - All undefined entries are made error.
  - Example: the LR parsing table above is an SLR(1) parsing table for the balanced-parentheses grammar.

4. DFA for Viable Prefixes

- A *viable prefix* is a prefix of a right sentential form that does not continue past the right end of the rightmost handle of that sentential form.
- The shift and goto functions of the canonical collection of sets of LR(0) items for a grammar *G* define a DFA that recognizes the viable prefixes of *G*.
- An item [A → β·γ] is *valid* for a viable prefix αβ if there is a rightmost derivation from S' to αAw to αβγw.

5. Practice Problems

Consider the following grammar G:

```
(1) S → S S +
(2) S → S S *
(3) S → a
```

- Construct a rightmost derivation and parse tree for the input string aaa*+$.
- Show the handle in each sentential form in the derivation.
- Construct the canonical collection of sets of LR(0) items for the augmented grammar.
- Construct an SLR(1) parsing table for G.
- Show how your SLR(1) parser processes the input string aaa*+$.

6. Reading

- ALSU, Sects. 4.5, 4.6.

---

aho@cs.columbia.edu

Homework Assignment #2
Submit solutions electronically on
Courseworks/COMSW4115/Assignments
by 2:40pm, March 6, 2013

Instructions

- Problems 1-4 are each worth 25 points.
- You may discuss the questions with others in the class but your answers must be in your own words and your own code. You must not copy someone else's solutions. If you consult others or use external sources, please cite the people or sources in your answers.
- Solutions to these problems will be posted on Courseworks on March 11, 2013.
- This assignment may submitted electronically on Courseworks by 2:40pm, March 11, 2013 for 50% credit.
- Pdf files are preferred.

Problems

1. Interactive desk calculator for boolean nor-expressions.
   a. Construct a grammar that generates boolean nor-expressions containing the logical constants `true` and `false`, the left-associative binary boolean operator `nor` [where `p nor q` means not (p or q)], and parentheses.
   b. Show the parse tree according to your grammar for the nor-expression

      ```
      true nor true nor (false nor false)
      ```

   c. Implement an interpreter that takes as input newline-terminated lines of boolean nor-expressions and produces as output the truth value of each expression. You can use lex and yacc or their equivalents to implement your interpreter. Show the source code for your interpreter and the sequences of commands you used to test it.
   d. Run your interpreter on the following two inputs and show the outputs:
      (a) `(true nor false) nor (true nor false)`
      (b) `true nor true nor (false nor false)`

2. Infix to stack machine code translator
   a. Consider the Yacc specification in Fig. 4.59 of ALSU (p. 292). Modify this translator to produce stack machine code for each input line. For example, for the input "1*(2+3)" your translator should produce the instructions

      ```
      push 1
      push 2
      push 3
      add
      multiply
      done
      ```

   b. Implement your translator in Yacc (or its equivalent) and show the stack machine code generated for each of the following inputs:
      (i) `1+2*3`
      (ii) `1+(2-3)`
      (iii) `1+2-+3`
      (iv) `1+2--3`

3. Let *L* be the language generated by the following grammar:

   $S \rightarrow a\ S\ b\ S \mid b\ S\ a\ S \mid \varepsilon$

   a. What language does this grammar generate?
   b. Show that this grammar is ambiguous.
   c. Construct the predictive parsing table for this grammar.
   d. Construct an LL(1) grammar for *L*.
   e. Construct the predictive parsing table for your grammar.

4. Let *L* be the language of pure lambda calculus expressions generated by the following grammar:

E → ^ v . E | E E | (E) | v

The symbols ^, ., (,) and v are tokens. ^ represents lambda and v represents a variable.

An expression of the form ^v.E is a function definition where v is the formal parameter of the function and E is its body.

If *f* and *g* are lambda expressions, then the lambda expression *fg* represents the application of the function *f* to the argument *g*.

a. Show that this grammar is ambiguous.
b. Construct an unambiguous grammar for *L* assuming that function application is left associative, e.g., *fgh* = (*fg*)*h*, and that function application binds tighter than ., e.g., (^*x*. ^*y*. *xy*) ^*z*.*z* = (^*x*. (^*y*. *xy*)) ^*z*.*z*.
c. Using your grammar, construct a parse tree for the expression (^*x*. ^*y*. *xy*) ^*z*.*z*.
d. Using the command yacc -v on a file containing your grammar, show the LALR(1) parsing action and goto table for your grammar.

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 11: Parsing Action Conflicts
February 27, 2013

Lecture Outline

1. Parsing action conflicts
2. Resolving shift/reduce conflicts
3. Using Yacc to generate LALR(1) parsers
4. Using Yacc with ambiguous grammars
5. Error recovery in Yacc

1. Parsing Action Conflicts

- If a grammar is not SLR(1), the SLR parsing table construction produces one or more multiply defined entries in the parsing table action function.
- These entries are either *shift/reduce conflicts* or *reduce/reduce conflicts.*

2. Resolving Shift/Reduce Conflicts

- **Example.** Given the augmented grammar *G'*

```
(0) E' → E
(1) E  → E+E
(2) E  → E*E
(3) E  → id
```

FOLLOW(E) = { +, *, $}

- Note that this grammar does not specify the relative precedence of the + and * operators nor their associativities.
- *C*, the canonical collection of sets of LR(0) items for *G'*, is

```
I0: E' → ·E
    E → ·E+E
    E → ·E*E
    E → ·id

I1: E' → E·
    E → E·+E
    E → E·*E

I2: E → id·

I3: E → E+·E
```

```
    E → ·E+E
    E → ·E*E
    E → ·id

I4: E → E*·E
    E → ·E+E
    E → ·E*E
    E → ·id

I5: E → E+E·
    E → E·+E
    E → E·*E

I6: E → E*E·
    E → E·+E
    E → E·*E
```

- SLR(1) parsing table for *G'*:

State	Action					Goto
	id	+	*	$	E
0	s2				1
1		s3	s4	acc	
2		r3	r3	r3	
3	s2				5
4	s2				6
5		s3 r1	s4 r1	r1	
6		s3 r2	s4 r2	r2	

- Notes
  - There is a shift/reduce conflict in `action[5,+]` between "`shift 3`" and "`reduce by E → E+E`" because the associativity of the operator + is not defined by the grammar. This conflict can be resolved in favor of "`reduce by E → E+E`" if we want + to be left associative.
  - There is a shift/reduce conflict in `action[5,*]` between "`shift 4`" and "`reduce by E → E+E`" because the relative precedence of the operators + and * is not defined by the grammar. This conflict can be resolved in favor of "`shift 4`" if we want * to have higher precedence than +.
  - Analogously, there is a shift/reduce conflict in `action[6,+]` between "`shift 3`" and "`reduce by E → E*E`" because the relative precedence of the operators + and * is not defined by the grammar. This conflict can be resolved in favor of "`reduce by E → E*E`" if we want * to have higher precedence than +.
  - There is a shift/reduce conflict in `action[6,*]` between "`shift 4`" and "`reduce by E → E*E`" because the associativity of the operator * is not defined by the grammar. This conflict can be resolved in favor of "`reduce by E → E*E`" if we want * to be left associative.

3. Using Yacc to Generate LALR(1) Parsers

- Consider the yacc program `expr1.y`:

```
%token id
%%
E : E '+' E
  | E '*' E
  | id
  ;
```

- Invoking `yacc -v expr1.y`, we can see the kernels of the sets of items for this grammar in the yacc output file `y.output`.
- The parsing action conflicts are shown for states 5 and 6.

```
state 0
       $accept : _E $end

       id  shift 2
       .  error
```

```
       E  goto 1

state 1
       $accept :  E_$end
       E :  E_+ E
       E :  E_* E

       $end  accept
       +  shift 3
       *  shift 4
       .  error

state 2
       E :  id_    (3)

       .  reduce 3

state 3
       E :  E +_E

       id  shift 2
       .  error

       E  goto 5

state 4
       E :  E *_E

       id  shift 2
       .  error

       E  goto 6

5: shift/reduce conflict (shift 3, red'n 1) on +
5: shift/reduce conflict (shift 4, red'n 1) on *
state 5
       E :  E_+ E
       E :  E + E_    (1)
       E :  E_* E

       +  shift 3
       *  shift 4
       .  reduce 1

6: shift/reduce conflict (shift 3, red'n 2) on +
6: shift/reduce conflict (shift 4, red'n 2) on *
state 6
       E :  E_+ E
       E :  E_* E
       E :  E * E_    (2)

       +  shift 3
       *  shift 4
       .  reduce 2


5/127 terminals, 1/600 nonterminals
4/300 grammar rules, 7/1000 states
4 shift/reduce, 0 reduce/reduce conflicts reported
4/601 working sets used
memory: states,etc. 17/2000, parser 2/4000
3/3001 distinct lookahead sets
0 extra closures
9 shift entries, 1 exceptions
3 goto entries
0 entries saved by goto default
Optimizer space used: input 25/2000, output 9/4000
9 table entries, 3 zero
```

```
maximum spread: 257, maximum offset: 257
```

4. Using Yacc with Ambiguous Grammars

- We can specify the associativities and relative precedence of the + and * operators in the declarations sections of a yacc program.
- Consider the yacc program `expr2.y`:
- The statement `%left '+'` makes + left associative.
- The statement `%left '*'` makes * left associative.
- Since the statement for + comes before the statement for *, + has lower precedence than *.

```
%token id
%left '+'
%left '*'
%%
E : E '+' E
 | E '*' E
 | id
 ;
```

- Invoking `yacc -v expr2.y`, we can now see that no shift-reduce conflicts have been generated in the yacc output file `y.output`.

```
state 0
       $accept : _E $end

       id  shift 2
       .  error

       E  goto 1

state 1
       $accept :  E_$end
       E :  E_+ E
       E :  E_* E

       $end  accept
       +  shift 3
       *  shift 4
       .  error

state 2
       E :  id_    (3)

       .  reduce 3

state 3
       E :  E +_E

       id  shift 2
       .  error

       E  goto 5

state 4
       E :  E *_E

       id  shift 2
       .  error

       E  goto 6

state 5
       E :  E_+ E
       E :  E + E_    (1)
       E :  E_* E
```

```
        *  shift 4
        .  reduce 1

state 6
        E :  E_+ E
        E :  E_* E
        E :  E * E_    (2)

        .  reduce 2


5/127 terminals, 1/600 nonterminals
4/300 grammar rules, 7/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
4/601 working sets used
memory: states,etc. 17/2000, parser 2/4000
3/3001 distinct lookahead sets
0 extra closures
6 shift entries, 1 exceptions
3 goto entries
0 entries saved by goto default
Optimizer space used: input 19/2000, output 10/4000
10 table entries, 3 zero
maximum spread: 257, maximum offset: 257
```

- Unless otherwise instructed, Yacc resolves all parsing action conflicts using the following two rules:
  1. A reduce/reduce conflict is resolved by choosing the conflicting production listed first in the Yacc specification.
  2. A shift/reduce conflict is resolved in favor of shift. Note that this rule correctly resolves the shift/reduce conflict arising from the dangling-else ambiguity.

5. Error Recovery in Yacc

- In Yacc, error recovery can be performed with error productions.
- To process errors at the level of a nonterminal A, add an error production $A \rightarrow$ `error` α where `error` is a Yacc reserved word, and α is a string of grammar symbols, possibly empty. Yacc will generate a parser including this production.
- If Yacc encounters an error during parsing, it pops symbols from its stack until it finds the topmost state on its stack whose underlying set of items includes an item of the form

  $A \rightarrow$. `error` α

  The parser then shifts the special token `error` onto the stack as though it saw the token `error` on the input.
- If α is not empty, Yacc skips ahead on the input looking for a substring that can be "reduced" to α on the stack. Now the parser will have `error` α on top of its stack, which it will reduce to A. The parser then resumes normal parsing.
- Example: See Fig. 4.61, p. 296. This Yacc specification contains a translation rule

```
lines : error '\n'  { yyerror("reenter previous line:"); yyerrok; }
```

On encountering an error, the parser starts popping symbols from its stack until it encounters a state with a shift action on `error`. The parser shifts the token `error` on to the stack and then skips ahead in the input until it finds a newline which it shifts onto the stack. The parser reduces `error '\n'` to `lines` and emits the error message "reenter previous line:". The special Yacc routine `yyerrok` resets the parser to its normal mode of operation.

6. Practice Problems

Consider the following grammar G:

(1) $S \rightarrow a\ S\ b\ S$
(2) $S \rightarrow b\ S\ a\ S$
(3) $S \rightarrow \varepsilon$

- What language does G generate?
- Construct an SLR(1) parsing table for G.
- Explain why the parsing action conflicts arise in the parsing table.
- Construct an equivalent LALR(1) grammar for L(G).

- Show that your grammar is LALR(1) by using yacc to construct an LALR(1) parsing table for it.
- Is your grammar LL(1)?

7. Reading

- ALSU, Sects. 4.8 and 4.9

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 12: Syntax-Directed Translation
March 4, 2013

Lecture Outline

1. The "dangling-else" ambiguity
2. Syntax-directed definitions and translation schemes
3. Synthesized and inherited attributes
4. S-attributed SDDs
5. L-attributed SDDs
6. Reading

1. The "Dangling-Else" Ambiguity

- Consider the following simplified ambiguous grammar for if- and if-else statements:

```
S' → S
S  → i S e S | i S | a
```

- Here the symbol `i` stands for `if expr then`, the symbol e stands for `else`, and the symbol a stands for all other productions. We have also added an augmenting production `S' → S`.
- The canonical collections of sets of LR(0) items for this grammar is as follows:

```
I0: S' → .S
    S  → .iSeS
    S  → .iS
    S  → .a

I1: S' → S.

I2: S  → i.SeS
    S  → i.S
    S  → .iSeS
    S  → .iS
    S  → .a

I3: S → a.

I4: S → iS.eS
    S → iS.

I5: S → iSe.S
    S → .iSeS
        S → .iS
        S → .a

I6: S → iSeS.
```

- The set of items I_4 gives rise to a shift/reduce conflict. The item `S → iS.eS` calls for a shift on e and since FOLLOW(S} = {e, $}, the item `S → iS.` calls for a reduction by production `S → iS.` on e. To associate each e with the closest unelsed if, we should resolve the conflict in favor of shift to state 5.
- See Section 4.8.2 of ALSU for a more detailed discussion.

2. Syntax-Directed Definitions and Translation Schemes

- The syntax analyzer translates its input token stream into an intermediate language representation of the source program, usually an abstract syntax tree (AST).
- A syntax-directed definition can be used to specify this translation.
- A syntax-directed definition (SDD) is a context-free grammar with attributes attached to grammar symbols and semantic rules attached to the productions.

- The semantic rules define values for attributes associated with the symbols of the productions. These values can be computed by creating a parse tree for the input and then making a sequence of passes over the parse tree, evaluating some or all of the rules on each pass. SDDs are useful for specifying translations.
- A syntax-directed translation scheme (SDTS) is a context-free grammar with program fragments, called semantic actions, embedded within production bodies. SDTSs are useful for implementing syntax-directed definitions.

3. Synthesized and Inherited Attributes

- Attributes are values computed at the nodes of a parse tree.
- *Synthesized attributes* are values that are computed at a node *N* in a parse tree from attribute values of the children of *N* and perhaps *N* itself. Synthesized attributes can be easily computed by a shift-reduce parser that keeps the values of the attributes on the parsing stack. See Sect. 5.4.2 of ALSU.
- An SDD is *S-attributed* if every attribute is synthesized. S-attributed SDDs are useful for bottom-up parsing.
- *Inherited attributes* are values that are computed at a node *N* in a parse tree from attribute values of the parent of *N*, the siblings of *N*, and *N* itself.
- An SDD is *L-attributed* is every attribute is either synthesized or inherited from the parent or from the left. L-attributed SDDs are useful for top-down parsing. See Sect. 5.2.4 of ALSU for details.

4. Examples of S-Attributed SDDs

- **Example 1.** Here is an S-attributed SDD translating signed bit strings into decimal numbers. The attributes, `BNum.val`, `Sign.val`, `List.val`, and `Bit.val`, are all synthesized attributes that represent integers.

```
BNum → Sign List      { BNum.val = Sign.val × List.val }
Sign → +              { Sign.val = +1 }
Sign → -              { Sign.val = -1 }
List → List1 Bit      { List.val = 2 × List1.val + Bit.val }
List → Bit            { List.val = Bit.val }
Bit  → 0              { Bit.val = 0 }
Bit  → 1              { Bit.val = 1 }
```

- **Example 2.** Here are Yacc translation rules implementing the SDD above for translating signed bit strings into decimal numbers. The identifiers $$, $1, $2 and so on in Yacc actions are synthesized attributes.

```
BNum : Sign List      { $$ = $1 * $2; }
     ;
Sign : '+'            { $$ = +1; }
     | '-'            { $$ = -1; }
     ;
List : List Bit       { $$ = 2*$1 + $2; }
     | Bit
     ;
Bit  : '0'            { $$ = 0; }
     | '1'            { $$ = 1; }
     ;
```

- **Example 3.** Here is an S-attributed SDD based on an SLR(1) grammar that translates arithmetic expressions into ASTs. `E` has the synthesized attributed `E.node` and `T` the synthesized attribute `T.node`. `E.node` and `T.node` point to a node in the AST. The function `Node(op, left, right)` returns a pointer to a node with three fields: the first labeled `op`, the second a pointer to a left subtree, and the third a pointer to a right subtree. The function `Leaf(op, value)` returns a pointer to a node with two fields: the first labeled `op`, the second the value of the token. See Example 5.11 in ALSU.

```
E → E1 + T     { E.node = Node('+', E1.node, T.node); }

E → T          { E.node = T.node; }

T → ( E )      { T.node = E.node; }

T → id         { T.node = Leaf(id, id.entry); }
```

5. Example of an L-Attributed SDD

- **Example 4.** Here is an L-attributed SDD based on an LL(1) grammar for translating arithmetic expressions into ASTs. See Example 5.12 in ALSU.

```
E → T A          { E.node = A.s;
                   A.i = T.node; }

A → + T A1       { A1.i = Node('+', A.i, T.node);
                   A.s = A1.s; }

A → ε            { A.s = A.i; }

T → ( E )        { T.node = E.node; }

T → id           { T.node = Leaf(id, id.entry); }
```

6. Practice Problems

1. Using Yacc, implement a syntax-directed translator that translates sequences of postfix Polish expressions into infix notation. For example, your translator should map `345+*` into `3*(4+5)`.
2. Optimize your translator so it doesn't generate any redundant parentheses. For example, your translator should still map `345+*` into `3*(4+5)` but it should map `345*+` into `3+4*5`.

7. Reading

- ALSU, Sects. 5.1-5.4

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 13: Intermediate Representations
March 6, 2013

Lecture Outline

1. Syntax-directed translation
2. Variants of syntax trees
3. Three-address code
4. Semantic analysis

1. Syntax-Directed Translation

- Postfix translation schemes
- Translation schemes with actions inside productions
- Producing ASTs with top-down parsing
  - Here is the L-attributed SDD based on an LL(1) grammar for translating arithmetic expressions into ASTs from Lecture 12.

```
E → T A          { E.node = A.s;
                   A.i = T.node; }

A → + T A1       { A1.i = Node('+', A.i, T.node);
                   A.s = A1.s; }

A → ε            { A.s = A.i; }

T → ( E )        { T.node = E.node; }

T → id           { T.node = Leaf(id, id.entry); }
```

2. Variants of Syntax Trees

- Abstract syntax trees
- Directed acyclic graphs
  - Algorithm 6.3: Value-number method for constructing a DAG (p. 361)

3. Three-Address Code

- Three-address instructions
- Representations for three-address code
  - Records
  - Quadruples
  - Triples
- Static single-assignment form

4. Semantic Analysis

- Uses made of semantic information for a variable x:
  - What kind of value is stored in x?
  - How big is x?
  - Who is responsible for allocating space for x?
  - Who is responsible for initializing x?
  - How long must the value of x be kept?
  - If x is a procedure, what kinds of arguments does it take and what kind of return value does it have?
- Storage layout for local names

5. Practice Problems

1. Construct a DAG for the expression
   `((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))`
2. Translate the following assignments into (a) syntax trees, (b) quadruples, (c) triples, (d) three-address code:
   a. `x = a + -(b+c)`
   b. `x[i] = y[i] + z[i]`
   c. `x = f(y+1) + 2`

6. Reading

- ALSU, Sections 6.1-6.3

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 14: Midterm Review
March 11, 2013

1. What you should know for the midterm

1. The different kinds of programming languages
   - Lecture 1
- The fundamental elements of programming languages
  - Lecture 2, ALSU Ch.1
- Language processing tools
  - Lecture 2, ALSU Ch. 1
- The structure of a compiler
  - Lecture 3, ALSU Chs. 1 and 2
- Regular languages, regular expressions, finite automata
  - Lectures 4 and 5, ALSU Ch. 3 except for Sect. 3.9

- Lexical analysis
  - Lectures 4 and 5, ALSU Chs. 2 and 3 except for Sect. 3.9
- Context-free languages and grammars
  - Lectures 6 and 7, ALSU Ch. 4 except for Sect. 4.7
- Top-down parsing
  - Lecture 9, ALSU Chs. 2 and 4 except for Sect. 4.7
- Bottom-up parsing
  - Lectures 10 and 11, ALSU Ch. 4 except for Sect. 4.7
- Syntax-directed translation
  - Lectures 12 and 13, ALSU Chs. 2 and Ch. 5 except for Sect. 5.5

2. Automata and Language Theory Review

- Regular languages
  - Finite automata
  - Regular expressions
  - Closure properties of regular languages
  - Decision properties of regular languages
  - Pumping lemma for regular languages and its uses
- Context-free languages
  - Context-free grammars
  - Parse trees, derivations, and ambiguity
  - Pushdown automata and deterministic pushdown automata
  - Closure properties of CFLs
  - Decision properties of CFLs
  - Pumping lemma for CFLs and its uses
- Syntax-directed translation
  - Syntax-directed definitions and translation schemes
  - Attribute grammars, inherited and synthesized attributes
  - S-attributed and L-attributed SDDs

3. Not all LL(1) grammars are SLR(1) and vice versa

- An LL(1) grammar that is not SLR(1)

$$S \rightarrow AaAb \mid BbBa$$
$$A \rightarrow \varepsilon$$
$$B \rightarrow \varepsilon$$

- An SLR(1) grammar that is not LL(1)

$$S \rightarrow SA \mid A$$
$$A \rightarrow a$$

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 15: Intermediate Code Generation
March 25, 2013

Lecture Outline

1. Intermediate Representations
2. Semantic analysis
3. Purpose of types in programming languages
4. Type systems
5. Typing in programming languages
6. Type inference rules

7. Type conversions

1. Intermediate Representations

- Abstract syntax trees
- Directed acyclic graphs (DAGs)
- Three-address code
  - Addresses
    - Names
    - Constants
    - Compiler-generated temporaries
  - Instructions
    - Three-address code has instructions of various forms for assignments, conditional and unconditional jumps, procedure calls and returns, indexed copy, address and pointer assignments.
  - Common representations for three-address code include records, quadruples, and triples.
- Static single-assignment (SSA) form is an intermediate reprentation that facilitates certain code optimizations. Assignments in SSA are to variables with distinct names. It also uses a special function, called a ϕ-function, to combine two definitions of the same variable arising from two different control-flow paths.

2. Semantic Analysis

- The semantic analyzer uses the syntax tree and information in the symbol table to check the source program for semantic consistency with the language definition.
- It gathers type information for intermediate code generation.
- Type checking is an important part of semantic analysis. During type checking the compiler checks that each operator has compatible operands.
- Uses made of semantic information for a variable `x`:
  - What kind of value is stored in `x`?
  - How big is `x`?
  - Who is responsible for allocating space for `x`?
  - Who is responsible for initializing `x`?
  - How long must the value of `x` be kept?
  - If `x` is a procedure, what kinds of arguments does it take and what kind of return value does it have?
- Storage layout for local names

3. Purpose of Types in Programming Languages

- Virtually all high-level programming languages associate types with values.
- Types often provide an implicit context for operations. For example, in C the expression `x + y` will use integer addition if `x` and `y` are `int`'s, and floating-point addition if `x` and `y` are `float`'s.
- Types can catch programming errors at compile time by making sure operators are applied to semantically valid operands. For example, a Java compiler will report an error if `x` and `y` are `String`'s in the expression `x * y`.

4. Type Systems

- The type of a construct in a program can be denoted by a type expression.
- A type expression is either a basic type (e.g., `integer`) or a type constructor applied to a type expression (e.g., a function from an integer to an integer).
- A type system is a set of rules for assigning type expressions to the syntactic constructs of a program and for specifying
  - type equivalence (when the types of two values are the same),
  - type compatibility (when a value of a given type can be used in a given context), and
  - type inference (rules that determine the type of a language construct based on how it is used).
- Forms of type equivalence
  - Name equivalence: two types are equivalent iff they have the same name.
  - Structural equivalence: two types are equivalent iff they have the same structure.
  - To test for structural equivalence, a compiler must encode the structure of a type in its representation. A tree (or type graph) is typically used.
- A type checker makes sure that a program obeys the type-compability rules of the language.
- We can think about types in several different ways:
  - Denotational: a type is a set of values called a domain.
  - Constructive: a type is either a primitive type (such as an integer or a character) or a composite type created by applying a type constructor (such as a structure or an array) to simpler types.

- Abstraction-based: a type is an interface consisting of a set of operations with well-defined and mutually consistent semantics.

6. Typing in Programming Languages

- The type system of a language determines whether type checking can be be performed at compile time (statically) or at run time (dynamically).
- A statically typed language is one in which all constructs of a language can be typed at compile type. C, ML, and Haskell are statically typed.
- A dynamically typed language is one in which some of the constructs of a language can only be typed at run time. Perl, Python, and Lisp are dynamically typed.
- A strongly typed language is one in which the compiler can guarantee that the programs it accepts will run without type errors. ML and Haskell are strongly typed.
- A type-safe language is one in which the only operations that can be performed on data in the language are those sanctioned by the type of the data. [Vijay Saraswat]

7. Type Inference Rules

- Type inference rules specify for each operator the mapping between the types of the operands and the type of the result.
- E.g., result types for `x + y`:

+	**int**	**float**
int	int	float
float	float	float

- Operator and function overloading
  - In Java the operator + can mean addition or string concatenation depending on the types of its operands.
  - We can choose between two versions of an overloaded function by looking at the types of their arguments.
- Function calls
  - Compiler must check that the type of each actual parameter is compatible with the type of the corresponding formal parameter. It must check that the type of the returned value is compatible with the type of the function.
  - The *type signature* of a function specifies the types of the formal parameters and the type of the return value.
  - Example: `strlen` in C
    - Function prototype in C:

```
unsigned int strlen(const char *s);
```

- Type expression:

```
strlen: const char * → unsigned int
```

- Polymorphic functions
  - A polymorphic function allows a function to manipulate data structures regardless of the types of the elements in the data structure
  - Example: Fig. 6.28 (p. 391) -- an ML program for the length of a list

8. Type Conversions

- Implicit type conversions
  - In an expression like `f + i` where `f` is a float and `i` is an integer a compiler must first convert the integer to a float before the floating point addition operation is performed. That is, the expression must be transformed into an intermediate representation like

```
t1 = INTTOFLOAT i
t2 = x FADD t1
```

- Explicit type conversions
  - In C, explicit type conversions can be forced ("coerced") in an expression using a unary operator called a cast. E.g., `sqrt((double) n)` converts the value of the integer `n` to a `double` before passing it on to the square root routine `sqrt`.

9. Practice Problems

The following grammar generates programs consisting of a sequence of declarations D followed by a single expression E. Each identifier must be declared before its use.

```
P → D ; E
D → D ; D | T id
T → int | float | T [ num ]
E → num | id | E [ E ] | E + E
```

- Construct type expressions as in Section 6.3.1 (pp. 371-372) for the following programs:
  a. int a; int b; a + b
  b. float[10][20] a; a[1] + a[2]
- Write pseudcode for a function `sequiv(exp1, exp2)` that will test the structural equivalence of two type expressions `exp1` and `exp2`.
- Show how your function computes `sequiv(array(2, array(2, int)), array(2, array(3, int)))`.

10. Reading

- ALSU, Sects. 6.1-6.4.

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 16: Translation of Statements
March 27, 2013

Lecture Outline

1. Logical rules of inference for type checking
2. Run-time storage organization
3. Translation of assignments
4. arrays
5. Boolean expressions
6. If-statements
7. While-statements

1. Logical Rules of Inference for Type Checking

- Type inference templates
  - We can specify the type inference rule "if expression `e1` has the type `int` and expression `e2` has the type `int`, then the expression `e1 + e2` has the type `int`" with a type inference template of the form

```
■ e1: int     ■ e2: int
_____________________

   ■ e1 + e2: int
```

- The turnstile symbol ■ is read "it is provable that" so the template can be read as "if it is provable that `e1` has type `int` and it is provable that `e2` has type `int`, then it is provable that `e1 + e2` has type `int`."
- Templates of this form provide a compact way of expressing the type rules of a language.
- We say a type system is sound if whenever ■ *e*: *T* then *e* evaluates to a value of type *T*.
- We can apply the type-inference templates by making a bottom-up traversal of the AST. We determine the types of the leaves using information from the symbol table. We can then move up the tree determining the type of the interior nodes from the types of their children by applying the inference rule for the operator at a given interior node.
- Type environments
  - A type environment is often needed to determine the type of a variable at a given node in the AST. A type environment is just a mapping from variables to types that is stored in the symbol table. The type environment for each node can be determined by making a top-down pass over the AST respecting the type scoping rules of the language.
  - The type inference rules are augmented with the type environment information. For example, if E is a type environment, we modify the template to make type inferences within the context of E:

```
E ■ e1: int     E ■ e2: int
_____________________________

   E ■ e1 + e2: int
```

- Static type checking can be done by making a top-down pass to compute the type environment for each node followed by a bottom-up pass to check the types at each node.

2. Run-time Storage Organization

- Run-time memory layout
  - Typical memory layout

```
(low address) Code
              Static data
              Runtime heap
                ↓

              Free memory

                ↑
(high address) Runtime stack
```

3. Translation of Assignments

- Here is Fig. 6.20 (p. 381), an SDTS generating three-address code for assignments:

```
S → id = E ; { gen(top.get(id.lexeme) '=' E.addr); }
E → E1 + E2  { E.addr = new Temp();
               gen(E.addr '=' E1.addr '+' E2.addr); }
 | - E1      { E.addr = new Temp();
               gen(E.addr '=' 'uminus' E1.addr); }
 | ( E1 )    { E.addr = E1.addr; }
 | id        { E.addr = top.get(id.lexeme); }
```

- Example. Here is the three-address code generated by this SDTS for the assignment statement: `a = b + -c;`

```
t1 = uminus c
t2 = b + t1
a = t2
```

4. Arrays

- Referencing a one-dimensional array
  - In C and Java, array elements are numbered `0, 1,..., n-1` for an array `A` with `n` elements.
  - Element `A[i]` begins in location `(base + i × w)` where `base` is the relative address of the storage allocated for `A` and `w` is the width of each element.
- Common layouts for multidimensional arrays
  - Row-major order
  - Column-major order
- See Fig. 6.22 (p. 383) for an SDD generating three-address code for assignments with array references.
- Example: three-address code for the expression `c + a[i][j]` assuming the width of an integer is 4

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a[t3]
t5 = c + t4
```

5. Boolean Expressions

- Boolean expressions are composed of boolean operators (&&, ||, !) applied to boolean variables, relational expressions, and other boolean expressions.
- Short-circuit evaluation: Some languages, such as C and Java, do not require an entire boolean expression to be evaluated.
  - Given `x && y`, if x is false, then we can conclude the entire expression is false without evaluating `y`.
  - Given `x || y`, if x is true, then we can conclude the entire expression is true without evaluating `y`.
- Numerical encoding
  - In C, the numerical value 0 represents false; a nonzero value represents true.
- Positional encoding
  - The value of a boolean expression can be represented by a position in three-address code, and the boolean operators can be translated into jumps.
  - The expression

```
if (x < 100 || x > 200 && x != y)
  x = 0;
```

can be translated into the following three-address instructions:

```
    if x < 100 goto L2
    ifFalse x > 200 goto L1
    ifFalse x != y goto L1
L2: x = 0
L1:
```

6. Translation of If-statements

- Boolean expressions often appear in the context of flow-of-control statements such as:
  - If statements
  - If-else statement
- See Figs. 6.36 (p. 402) and 6.37 for SDDs translating these statements with booleans into three-address code.
- For the expression

```
if (x < 100) || x > 200 && x != y)
  x = 0;
```

these SDDs produce the following three-address instructions:

```
    if x < 100 goto L2
    goto L3
L3: if x > 200 goto L4
    goto L1
L4: if x != y goto L2
    goto L1
L2: x = 0
L1:
```

This code can be transformed into the code in Section 5 by eliminating the redundant goto and changing the directions of the tests in the second and third if-statements.

7. Translation of While-statements

- Consider the production `S` $\rightarrow$ `while ( B ) S1` for while-statements. The shape of the code for implementing this production can take the form:

```
begin: // beginning of code for S
   code to evaluate B
   if B is true goto B.true
   if B is false goto B.false
B.true:
   code to evaluate S1
```

```
   goto begin
B.false:  // this is where control flow will go after executing S
```

- Here is an SDD for this translation (from Fig. 6.36, p. 402):

```
S → while ( B ) S1 {
         begin = newlabel()
         B.true = newlabel()
         B.false = S.next
         S1.next = begin
         S.code = label(begin) || B.code ||
                  label(B.true) || S1.code ||
                  gen('goto' begin)
}
```

8. Practice Problems

1. Use the SDD of Fig. 6.22 (ALSU, p. 383) to translate the assignment `x = a[i][j] + b[i][j]`.
2. Add rules to the SDD in Fig. 6.36 (ALSU, p. 402) to translate do-while statements of the form:

   $S \rightarrow$ **do** S **while** B

   Show the code your SDD would generate for the program

```
do
  do
    assign1
  while a < b
while c < d
```

9. Reading

- ALSU, Sections 6.4 - 6.8, 7.1

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 17: Procedures
April 1, 2013

Lecture Outline

1. Names
2. Procedures
3. Parameter-passing mechanisms
4. Evaluation strategies
5. Storage-allocation strategies
6. Activation trees and records

1. Names

- A name is a character string used to represent something.
- Names in most languages are identifiers.
- In some programming languages, certain identifiers fall into distinct namespaces that do not interfere with one another. For example, in the C structure declaration

```
struct id {  /* here id is a structure tag */
```

```
 int id;    /* here id is a structure member */
} id;        /* here id is a structure variable */
```

the structure tag `id`, the structure member `id`, and the structure variable `id` are in different namespaces and hence distinct identifiers that can be distinguished by context.

- A binding is an association between two things such as between a name and its type or between a symbol and the operation it represents. The time at which this association is determined is called the binding time. Bindings can take place from language design time to runtime.
- The textual region of a program in which a binding is active is called its scope. A scope is often with respect to a namespace.
  - In a language with static scoping, the bindings between names and objects are determined at compile time by examining the text of the program. Static scoping is sometimes called lexical scoping. In static scoping, a name refers to its lexically closest declaration.
  - In a language with dynamic scoping, the bindings between names and objects depend on the order in which procedures are called at runtime.
- Lifetimes
  - The lifetime of a name-object binding is the time between the creation and destruction of that binding.
  - The lifetime of an object is the time between the creation and destruction of the object. Depending on the language, the lifetime of an object may be different than that of lifetime of the name-object binding.
    - In C, an object is a location in storage. The storage class determines the lifetime of the storage associated with an object.
    - In C, there are two storage classes: automatic and static. Automatic objects are local to a block and are discarded on exit from the block. Static objects retain their values across entry from and reentry to functions and blocks.
  - Object lifetimes are usually determined by the storage-allocation strategy used to manage the storage for that object.
    - Static objects are allocated memory in the code space and have an address that is retained throughout the execution of a program.
    - Stack objects are allocated in a last-in, first-out order on the runtime stack usually in conjunction with procedure calls and returns.
    - Heap objects are dynamically allocated and deallocated at arbitrary times on the runtime heap. Some languages such as Java and C# use a garbage collection mechanism to identify and reclaim heap objects that become unreachable during program execution.

2. Procedures

- A procedure *P* in a programming language is a collection of statements that defines a parameterized computation. An invocation of *P* is called an activation of *P*.
- We use the term *actual parameters* to denote the parameters used in the call of a procedure.
- We use the term *formal parameters* to denote the parameters used in the definition of a procedure.
- We will often call a procedure that returns a value a *function*. (C uses the term function for procedure as well.)
- The type of the function `return_type f(arg1_type a, arg2_type b)` can be denoted by the type expression

  `arg1_type x arg2_type` $\rightarrow$ `return_type`
- Some design issues for implementing procedures
  - choice of parameter-passing mechanism
  - storage allocation for local variables: static or dynamic
  - can procedure declarations nest
  - can procedures be passed as parameters, returned as values
  - can procedure names be overloaded
  - generic procedures, ones whose computations can be done on different types
  - does language have closures (encapsulations of procedures with their runtime context)

3. Parameter-Passing Mechanisms

- Programming languages differ in how the values of parameters are passed to called procedures.
- Call by value
  - The actual parameter is evaluated if it is an expression or copied if it is a variable. The r-value is placed in the location belonging to the corresponding formal parameter of the called procedure.
  - C and Java use call by value. C leaves the order in which the parameters are evaluated unspecified; Java evaluates the parameters left to right.
  - "swap" example from C
    - Consider the following C program fragment

```
a = 1;
b = 2;
swap(a, b);
printf("a = %d, b = %d\n", a, b);
```

where the function `swap` is defined as

```
void swap(int x, int y) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

- Now consider the same program fragment with `swap(&a, &b)` in place of `swap(a, b)` and with `swap` defined as

```
void swap(int *px, int *py) {
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

- Call by reference
  - The address of the actual parameter is passed to the callee as the value of the corresponding formal parameter.
  - If the parameter is an expression, the expression is evaluated and its value is stored in a new location before the call. The address of that location is passed.
  - Useful for passing large parameters to procedures.
  - Used for reference parameters in C++. In C++, `swap` can be written with reference parameters as

```
void swap(int &x, int &y) {
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

  In the body, `x` and `y` are `int`'s, not pointers to `int`'s. The caller passses as parameters the variables whose values are to be swapped, not their addresses.
- Call by name
  - A call-by-name parameter is re-evaluated in the caller's referencing environment each time it is used. The effect is as though the called procedure is textually expanded at the point of the call with each actual parameter substituted for the corresponding formal parameter at every occurrence in the body of the procedure. Local names in the called procedure may need to be renamed to keep them distinct.
  - Used in Algol 60.
  - Also used at compile time by macros in the C preprocessor.
  - Example: Consider the macro definition in C

    ```
    #define max(a, b) ((a) > (b) ? (a) : (b))
    ```

    The C statement

    ```
    x = max(p+q, r*s);
    ```

    will be replaced by the statement

    ```
    x = ((p+q) > (r*s) ? (p+q) : (r*s);
    ```

4. Evaluation Strategies for the Arguments of a Procedure

- An evaluation strategy defines when and in what order the parameters to a procedure are evaluated.
- In applicative-order evaluation, all parameters are evaluated before applying the procedure. C functions and Java methods use applicative-order evaluation.
- In normal-order evaluation, parameters are evaluated after applying the procedure, and then only if the result is needed to complete the evaluation of the procedure. Normal-order evaluation is used with macros and call-by-name parameters. Haskell uses a memoized version of call by name called call by need.

5. Storage-Allocation Strategies

- Static allocation
  - Storage for all data objects is laid out at compile time.
  - Names are bound to storage as program is compiled.

- Static allocation was used in early versions of Fortran.
- Recursion is restricted.
- Size of all data objects must be known at compile time.
- No dynamic data structures can be supported.

- Stack allocation
  - Run-time storage is organized as a stack.
  - Activation records (ARs) are pushed and popped as activations of procedures begin and end.
  - Typical kinds of data appearing in an activation record:

```
Actual parameters
Returned values
Control link
Access link
Saved machine status
Local data
Temporaries
```

- Storage for the locals in each call is contained in the AR for that call.
- Used by C and Java.
- Heap allocation
  - Storage is allocated and deallocated as needed at run time from a data area called a heap.
  - Necessary when data outlives the call to the procedure that created it.
  - Also needed when the values of local names must be retained after an activation ends.

6. Activation Trees and Records

- Consider the following C program for Euclid's algorithm.

```
#include
int x, y;
int gcd(int u, int v) {
  if (v == 0)
     return u;
  else
     return gcd(v, u%v);
}

main() {
  scanf("%d%d", &x, &y);
  printf("%d\n", gcd(x, y));
  return 0;
}
```

- A tree, called an activation tree, can be used to represent the procedure calls made during an execution of gcd because the lifetimes of the procedure activations are nested.
- Note that:
  - The sequence of procedure calls corresponds to a preorder traversal of the tree.
  - The sequence of returns corresponds to a postorder travseral.
  - The path from the root to a node *N* shows the activations that are live at the time *N* is executing.
- Procedure calls and returns are managed by a control stack.
- On each procedure call, an activation record for that procedure is pushed on the stack. The activation record for each procedure call contains the information needed to manage the execution of that procedure call. When the call returns, that activation record is popped from the stack.

7. Reading

- ALSU, Sections 6.9, 7.1, 7.2

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Homework Assignment #3
Submit solutions electronically on
Courseworks/COMSW4115/Assignments
by 2:40pm, April 10, 2013

Instructions

- Problems 1-4 are each worth 25 points.
- You may discuss the questions with the TAs and others in the class but your answers must be in your own words. You must not copy someone else's solutions. If you consult others or use external sources, please cite the people or sources used in your answers.
- Solutions to these problems will be posted on Courseworks on April 17, 2013.
- This assignment may submitted electronically on Courseworks by 2:40pm, April 17, 2013 for 50% credit.
- Pdf files are preferred.

Problems

1. Consider the syntax-directed definitions in Figs. 6.19, 6.36 and 6.37 in ALSU for expressions, if-statements and booleans. Assume the boolean operators `&&`, `||`, and `!=` have the customary associativities and precedences.
   a. Construct a parse tree for the C-like if-statement
      `if( x < 10 && x > 20 || x != y ) x = 30;`
   b. Show the values of all the attributes computed at each node in the parse tree by these SDDs.
   c. Show the three-address code produced for this if-statement.
   d. Can you see any ways in which the three-address code can be optimized?

2. Write pseudcode for a function `sequiv(exp1, exp2)` that will test the structural equivalence of two type expressions `exp1` and `exp2`. Show how your function computes `sequiv(array(2, array(2, int)), array(2, array(3, int)))`.

3. Let `fib(n)` be the function

```
int fib(n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

   a. Show the activation tree for `fib(3)`.
   b. Show the activation records that are on the run-time stack when `fib(1)` is invoked for the first time. Just show the return value, actual parameters, and the caller's frame pointer in each activation record.

4. Give an example from some common programming languages to illustrate the difference between
   a. Normal-order evaluation and applicative-order evaluation.
   b. Call by reference and call by value.
   c. Lexical scope and dynamic scope.
   d. Stack allocation and heap allocation.
   e. A static type and a dynamic type.

   You can use different programming languages for each part.

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 18: Run-time Environments
April 3, 2013

Lecture Outline

1. Activation trees
2. Calling sequences
3. Access to local arrays
4. Heap memory management

1. Activation Trees

- Consider the following C program:

```
int x = 2;

void f(int n) {
  static int x = 1;
  g(n);
  x--;
}

void g(int m) {
  int y = m-1;
  if (y > 0) {
    f(y);
    x--;
    g(y);
  }
}

main() {
  g(x);
  return 0;
}
```

- The activation tree for this program is

```
   main()
     |
    g(2)
    / \
 f(1) g(1)
  |
 g(1)
```

2. Calling Sequences

- Procedure calls are implemented by calling sequences, code that allocates an activation record on the control stack and enters information into its fields.
- A return sequence is code invoked after the call to restore the state of the machine so the calling procedure can continue its execution after the call.
- The code in a calling sequence is usually divided between the calling procedure (the "caller") and the procedure it calls (the "callee").
- When a procedure p calls a procedure q, we might do something like the following:
  - Evaluate the parameters of q and store them in a new AR for q.
  - Store the frame pointer (fp) for p as the control link in the AR for q.
  - Update fp to point to the AR of q.
  - Store the return address in the AR for q.
  - Jump to the code for q.

    - Have q allocate and initialize its local data and temporaries on the stack.
- When q exits:
    - Copy the fp of the AR for q into sp (the top of stack pointer).
    - Load the control link of the AR for q into the fp.
    - Jump to the return address.
    - Change the stack pointer to pop the parameters of q off the run-time stack.
- Contrast the run-time stack during the first and second calls to `g(1)` in the program in section (1).

3. Allocating Space for Arrays

- Fixed-size arrays

```
void f(int x) {
  int a;
  int b[4];
  int c;
  ...
}
```

- Space for the array b can be allocated on the runtime stack between the space for a and c.
- Variable-size arrays

```
void f(int x) {
  int a;
  int b[n];
  int c;
  ...
}
```

- A pointer to the space for b can be allocated on the runtime stack between the space for a and c so variables remain a constant offset from the frame pointer. The actual space for b can be allocated after c.

4. Heap Memory Management

- The runtime heap is used for data objects whose lifetimes may exist long after the procedure that created them.
- The heap memory manager is the subsystem that allocates and deallocates space within the heap.
- Garbage collection is the process of finding memory within the heap that is no longer used by the program and can therefore be reallocated to house other data objects. In some programming languages like C allocatation and deallocation needs to be done manually using library functions like `malloc` and `free`. In other languages like Java it is the garbage collector that deallocates memory.
- There are many different garbage collection algorithms that vary in performance metrics such as execution time, space usage, pause time, and program locality.
- Mark-and-sweep garbage collection
    - A basic mark-and-sweep (M&S) garbage collection algorithm is straightforward to implement and works well for languages like C because it does not move data objects during garbage collection.
    - A M&S collector firsts visits and marks all reachable data objects in the heap, setting the reached-bit of each object to 1.
    - It then sweeps the entire heap, freeing up unreachable objects.
    - See Algorithm 7.12 (ALSU, p. 471) for details.

5. Practice Problem

- Consider the following program written in a hypothetical statically scoped language that allows nested functions. In this program, main calls f which calls g2 which calls h which calls g1.

```
function main() {
  int i;

  function f() {
    int a, b, c;

    function g1() {
      int a, d;
```

```
      a = b + c;              // point 1

    }; // end of g1

    function g2(int x) {
      int b, e;

      function h() {
        int c, e;

        g1();
        e = b + a;            // point 2

      }; // end of h

      h();
      a = d + e;              // point 3

    }; // end of g2

    g2(1);

  }; //end of f

  // execution of main begins here

  f();

}; // end of main
```

a. Suppose we have activation records with the following fields:
   Parameters
   Control Link
   Access Link
   Return Address
   Local data

If function p is nested immediately within function q, then the access link in any AR for p points to the most recent AR for q.

Show the activation records on the run-time stack when execution first arrives at point 1 in the program above.

- To which declarations are the references to variables a, b, c at position 1?
- To which declarations are the references to variables a, b, e at position 2?
- To which declarations are the references to variables a, d, e at position 3?

6. Reading

- ALSU, Sections 7.1, 7.2, 7.6.1
- K. C. Louden, Compiler Construction, PWS Publishing, 1997.

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 19: Code Generation
April 8, 2013

Lecture Outline

1. Issues in code generation
2. Memory hierarchy
3. Categories of target machines
4. Primary tasks of a code generator

5. Basic blocks and flow graphs

1. Issues in Code Generation

- Role of the code generator
  - Input: IR of the source program produced by the font end
  - Output: good target machine code
- Challenges of code generation
  - The target program must preserve the semantic meaning of the source program.
  - The target program should make efficient use of the target machine's resources.
  - The code generator itself should be efficient.
  - The problem of optimal code generation is undecidable.
  - Many subproblems in code generation, such as optimal register allocation, are computationally intractable.
  - The design of a good code generator often reverts to the problem of designing good heuristics.

2. Memory Hierarchy

- Processor registers are the fastest devices in a computer's memory hierarchy. Here are some typical access times and sizes of the components of a computer's memory hierarchy.

```
               Access Time                  Size
registers       0.2 - 0.5 ns              256 -  1024 B
L1 cache       0.4 -   1 ns               32 -   256 KB
L2 cache         4 -  10 ns              512 KB -  2 MB
main memory        50 - 500 ns              256 MB - 16 GB
disk                 5 -  15 ms               80 GB
tape                 1 -  50 s              infinite
```

- As we can see, there are several orders of magnitude difference in the access time between registers and main memory. As a consequence, compiler code generators attempt to use registers to hold the most frequently used objects in the target program. But since there are a small number of registers, the compiler will have to make choices as to what quantities it keeps in registers and what quantities it must spill into main memory.

3. Categories of Target Machines

- Reduced instruction set machines (RISC)
  - many registers
  - three-address instructions
  - simple addressing modes
  - simple instruction set architecture
- Complex instruction set machines (CISC)
  - few registers
  - two-address instructions
  - variety of addressing modes
  - several register classes
  - variable-length instructions
  - instructions with side effects
- Stack-based machines
  - push operands onto stack
  - perform operations on operands at top of stack
  - stack kept in registers
  - model for Java Virtual Machine
- Multicore machines

4. Primary Tasks of a Code Generator

- Instruction selection
  - Determining factors:
    - level of IR
    - nature of ISA
    - desired quality of generated code

- Register allocation and assignment
  - Register allocation determines the set of variables that will reside in registers at each point in the program.
  - Register assignment determines the specific register in which a variable will reside.
- Evaluation order
  - Some computation orders require fewer registers to hold intermediate results than others.
  - Picking an optimal order in the general case is NP-complete.

5. Basic Blocks and Flow Graphs

- A basic block is a maximal sequence of consecutive three-address instructions such that

1. The flow of control can only enter the basic block throught the first instruction in the block.
2. Control will leave the block without halting or branching except possibly at the last instruction in the block.

- A flow graph for the basic blocks of an intermediate program can be constructed as follows:
  - The basic blocks are the nodes of the flow graph.
  - There is an edge from block B to block C iff it is possible for the first instruction in C to immediately follow the last instruction in B.
- A set of nodes L in a flow graph is a loop if
  1. There is a node in L called the loop entry with the property that no other node in L has a predecessor outside L.
  2. Every node in L has a nonempty path completely within L to the entry of L.

6. Practice Problem

1. ALSU, Exercise 8.4.1, p. 531.

7. Reading

- ALSU, Sections 8.1-8.5

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 20: Code Generation Algorithms
April 10, 2013

1. Target Machine

- *n* general-purpose registers
- Instructions: load, store, compute, jump, conditional jump
- Various addressing modes:
  - indexed address
  - integer indexed by a register
  - indirect addressing
  - immediate constant
- Example 1: for `x = y + z` we can generate

```
LD  R1, y       // R1 = y
LD  R2, z       // R2 = z
ADD R1, R1, R2  // R1 = R1 + R2
ST  x, R1       // x  = R1
```

- Example 2: for `b = a[i]` where a is an array of integers we can generate

```
LD  R1, i       // R1 = i
MUL R1, R1, #4  // R1 = R1 * 4
LD  R2, a(R1)   // R2 = contents(a + contents(R1))
ST  b, R2       // b  = R2
```

- Example 3: for `a[j] = c` where a is an array of integers we can generate

```
LD  R1, c       // R1 = c
LD  R2, j       // R2 = j
MUL R2, R2, #4  // R2 = R2 * 4
ST  a(R2), R1   // contents(a + contents(R2)) = R1
```

- Example 4: for `x = *p` we can generate

```
LD  R1, p       // R1 = p
LD  R2, 0(R1)   // R2 = contents(0 + contents(R1))
ST  x, R2       // x = R2
```

- Example 5: for `*p = y` we can generate

```
LD  R1, p       // R1 = p
LD  R2, y       // R2 = y
ST  0(R1), R2   // contents(0 + contents(R1)) = R2
```

- Example 6: for `if x < y goto L` we can generate

```
LD   R1, x      // R1 = x
LD   R2, y      // R2 = y
SUB  R1, R1, R2 // R1 = R1 - R2
BLTZ R1, M      // if R1 < 0 jump to M
```

  Here `M` is the label of the first machine instruction generated from the three-address instruction that has label `L`
- Instruction cost: 1 + cost associated with the addressing modes of the operands

2. Names to Addresses

- Addresses in the target code are in the runtime address space.
- Names in the IR need to be converted into addresses in the target code.
- Example: managing the addresses in the runtime stack
  - The code for the first procedure initializes the runtime stack by setting the stack pointer SP to the start of the stack.
  - A procedure call increments SP, saves the return address, and transfers control to the called procedure.
  - In the return sequence
    - the called procedure transfers control to the return address using the instruction `BR *0(SP)`
    - the caller decrements SP to its previous value

3. Computing Next-Use Information

- Knowing when the value of a variable will be used next is essential for generating good code.
- If there is a three-address instruction sequence of the form

```
i:  x = y + z

.
. no assignments to x between instructions i and j
.

j:  a = x + b
```

  then we say statement `j` *uses* the value of `x` computed at `i`.
- We also say that variable `x` is *live* at statement `i`.
- A simple way to find next uses is to scan backward from the end of a basic block keeping track for each name `x` whether `x` has a next use in the block and if not whether `x` is live on exit from that block. See Alg. 8.7, p. 528.

4. A Simple Code Generator

- Here we describe an algorithm for generating code for a basic block that keeps track of what values are in registers so it can avoid generating unnecessary loads and stores.
- It uses a register descriptor to keep track of what variable values are in each available register.
- It uses an address descriptor to keep track of the location or locations where the current value of each variable can be found.
- For the instruction x = y + z it generates code as follows:
  - It calls a function getReg(x = y + z) to select registers Rx, Ry, and Rz for variables x, y, and z.
  - If y is not in Ry, it issues the load instruction `LD Ry, My` where My is one of the memory locations for y in the address descriptor.
  - Similarly, if z is not in Rz, it issues a load instruction `LD Rz, Mz`.
  - It then issues the instruction `ADD Rx, Ry, Rz`.
- For the instruction x = y it generates code as follows:
  - It calls a function getReg(x = y) to select a register Ry for both x and y. We assume retReg will always choose the same register for both x and y.
  - If y is not in Ry, issue the load instruction `LD Ry, My` where My is one of the memory locations for y in the address descriptor.
  - If y is already in Ry, we issue no instruction.
- At the end of the basic block, it issues a store instruction `ST x, R` for every variable x that is live on exit from the block and whose current value resides only in a register R.
- The register and address descriptors are updated appropriately as each machine instruction is issued.
- If there are no empty registers and a register is needed, the function getReg generates a store instruction `ST v, R` to store the value of the variable v in some occupied register R. Such a store is called a *spill*. There are a number of heuristics to choose the register to spill.

5. Optimal Code Generation for Expression Trees

- In this section we assume we are using a *k*-register machine with instructions of the form
  - `LD reg, mem`
  - `ST mem, reg`
  - `OP reg, reg, reg`

  to evaluate expressions.
- Ershov numbers
  - An expression tree is a syntax tree for an expression.
  - Numbers, called Ershov numbers, can be assigned to label the nodes of an expression tree. The Ershov number at a node gives the minimum number of registers needed to evaluate on a register machine the expression generated by that node with no spills.
  - Algorithm to label the nodes of an expression tree with Ershov numbers

  1. Label all leaves 1.
  2. The label of an interior node with one child is the label of its child.
  3. The label of an interior node with two children is the larger of the labels of its children if these labels are different; otherwise, it is one plus the label of the left child.
- Sethi-Ullman algorithm generates register machine code that minimizes the number of spills to evaluate an expression tree. Ershov numbers guide the evaluation order.
  - Input: an expression tree labeled with Ershov and a *k*-register machine.
  - Output: an optimal sequence of register machine instructions to evaluate the root of the tree into a register.
  - The details of the algorithm are in Section 8.10 of ALSU, pp. 567-573.

6. Practice Problems

1. ALSU, Exercise 8.2.5 (p. 517).
2. ALSU, Exercise 8.10.2 (p. 573).

7. Reading

- ALSU, Sections 8.2-8.4, 8.6, 8.10

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Homework Assignment #4

Submit solutions electronically on
Courseworks/COMSW4115/Assignments
by 2:40pm, April 24, 2013

Instructions

- Problems 1-4 are each worth 25 points.
- You may discuss the questions with the TAs and others in the class but your answers must be in your own words. You must not copy someone else's solutions. If you consult others or use external sources, please cite the people or sources used in your answers.
- Solutions to these problems will be posted on Courseworks on May 1, 2013.
- This assignment may submitted electronically on Courseworks by 2:40pm, May 1, 2013 for 50% credit.
- Pdf files are preferred.

Problems

1. Consider the arithmetic expression `u * (v - w) + x / y` and a register machine with instructions of the form

```
LD reg, src
ST dst, reg
OP reg1, reg2, reg3      // the registers need not be distinct
```

   a. Draw an abstract syntax tree for the expression and label the nodes with Ershov numbers.
   b. Generate machine code for the expression on a two-register machine minimizing the number of spills.

2. Consider the following sequence of three-address code:

```
  x = 0
  i = 0
L: t1 = i * 4
  t2 = a[t1]
  t3 = i * 4
  t4 = b[t3]
  t5 = t2 * t4
  x = x + t5
  i = i + 1
  if i < n goto L
```

   a. Draw a flow graph for this three-address code.
   b. Optimize this code by eliminating common subexpressions, performing reduction in strength on induction variables, and eliminating all the induction variables that you can. State what transformations you are using at each optimization step.

3. Consider the lambda-calculus expression (λu. (λx. u) u) ((λy. y) (λw. (λv.v) w)).
   a. Draw a parse tree for this expression.
   b. Identify all redexes in this expression.
   c. Evaluate this expression using normal order evaluation.
   d. Evaluate this expression using applicative order evaluation.

4. Evaluate the lambda-calculus expression `(λx.(λy.(x(λx.xy))))y`. What order of evaluation did you use? Explain all the steps you used in the evaluation.

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 22: April 17, 2013

Introduction to Lambda Calculus

Outline

- The evolution of programming languages
- Programming paradigms
- History of lambda calculus and functional programming languages
- CFG for lambda calculus
- Function abstraction
- Function application
- Free and bound variables
- Beta reductions
- Evaluating a lambda expression
- Currying
- Renaming bound variables by alpha reduction
- Eta conversion
- Substitutions
- Disambiguating lambda expressions
- Normal form
- Evaluation strategies

1. The Evolution of Programming Languages

- There are thousands of programming languages in the world today. Why so many?
  - Software is used in virtually every domain of human endeavor. Each application domain has its distinct and often conflicting programming needs. No one language is ideal for all application domains.
    - Scientific programming demands support for matrices and efficient floating point computations. Fortran, first developed in the 1950s, is still widely used for scientific computation.
    - Systems programming demands low-level control of machine resources, often with real-time constraints. C and C++ are the dominant systems programming languages.
    - Business applications are characterized by data persistence and report generation. SQL is the dominant query language in business data processing.
- Why are there so many new programming languages?
  - Software reliability and programmer productivity are critical concerns.
  - Programmer training is a significant cost, so old languages tend to persist.
  - But new languages can arise to meet the demands of new application domains. For example, Java was designed in the 1990s to meet the demands of Internet programming. But to gain acceptance, Java was designed to look a lot like C++.
- What makes a good programming language?
  - This is a hotly debated question. There is no universally accepted metric for the goodness of a programming language. Technical excellence is rarely a criterion for adoption.
  - Functional programming languages have many ardent advocates and many features of functional languages have been incorporated into modern languages. The next sequence of lectures will discuss the theoretical underpinnings of functional languages and highlight a few important functional languages.

2. Programming Paradigms

- A programming paradigm is a style of computer programming.
- There are many paradigms, each with its distinctive characteristics and prototypical programming languages. The four most common are:
  - imperative also known as procedural programming
    - imperative programming tends to be the most popular paradigm
    - an imperative program tells the computer *how* to do a computation
    - the typical imperative languages are C and Fortran
  - object-oriented programming
    - an object-oriented program consists of a collection of interacting objects
    - C++, Java, and Smalltalk are prototypical object-oriented languages
  - functional programming
    - a functional program is typified by the recursive definition of functions
    - Haskell, Lisp, Scheme, ML, and OCaml are popular functional languages
  - declarative programming
    - a declarative program specifies *what* should be computed

    - SQL and Prolog are declarative languages
- Some programming languages are multi-paradigm languages in the sense that they can support more than one programming paradigm. For example, C#, OCaml, Ruby, and Scala support both imperative, object-oriented, and functional programming.

3. History of Lambda Calculus and Functional Programming Languages

- Lambda calculus was introduced in the 1930s by Alonzo Church at Princeton University as a mathematical system for defining computable functions.
- Lambda calculus is equivalent in definitional power to that of Turing machines.
- Lambda calculus serves as the theoretical model for functional programming languages and has applications to artificial intelligence, proof systems, and logic.
- Lisp was developed by John McCarthy at MIT in 1958 around lambda calculus.
- ML, a general-purpose functional language, was developed by Robin Milner at the University of Edinburgh in the early 1970s. Caml and OCaml are dialects of ML developed at INRIA in 1985 and 1996, respectively.
- Haskell, considered by many as one of the purest functional programming languages, was developed by Simon Peyton Jones, Paul Houdak, Phil Wadler and others in the late 1980s and early 90s.
- Because of its simplicity, lambda calculus is a very useful tool for the study and analysis of programming languages.

4. CFG for The Lambda Calculus

- The central concept in lambda calculus is an expression which we can think of as a program that when evaluated returns a result consisting of another lambda calculus expression.
- Here is the grammar for lambda expressions:

```
expr → λ variable . expr | expr expr | variable | ( expr ) | built_in
```

- A `variable` is an identifier.
- A `built_in` is a built-in function such as addition or multiplication, or a constant such as an integer or boolean. However, as we shall see, all programming language construct can be implemented as functions with the pure lambda calculus so these built-ins are unnecessary. However, we will use built-ins for notational simplicity.

5. Function Abstraction

- A function abstraction, often called a lambda abstraction, is a lambda expression that defines a function.
- A function abstraction consists of four parts: a lambda followed by a variable, a period, and then an expression as in λ`x.expr`.
- In the function abstraction λ`x.expr` the variable `x` is the formal parameter of the function and `expr` is the body of the function.
- For example, the function abstraction λ*x*. + *x* 1 is a function of *x* that adds *x* to 1. Parentheses can be added to lambda expressions for clarity. Thus, we could have written this function abstraction as λ*x*.(+ *x* 1) or even as (λ*x*. (+ *x* 1)).
- In C this function definition might be written as

```
int addOne (int x)
{
 return (x + 1);
}
```

- Note that unlike C the lambda abstraction does not give a name to the function. The lambda expression itself is the function.
- We say that λ`x.expr` *binds* the variable `x` in `expr` and that `expr` is the *scope* of the variable.

6. Function Application

- A function application, often called a lambda application, consists of an expression followed by an expression: `expr expr`. The first expression is a function abstraction and the second expression is an argument to which the function is applied.
- For example, the lambda expression λ*x*. (+ *x* 1) 2 is an application of the function λ*x*. (+ *x* 1) to the argument 2.
- This function application λ*x*. (+ *x* 1) 2 can be evaluated by substituting the argument 2 for the formal parameter *x* in the body (+ *x* 1). Doing this we get (+ 2 1). This substitution is called a beta reduction.
- Beta reductions are like macro substitutions in C. To do beta reductions correctly we may need to rename bound variables in lambda expressions to avoid name clashes.
- Functions can be used as arguments to functions and functions can return functions as results.

7. Free and Bound Variables

- In the function definition λ*x*.*x* the variable *x* in the body of the definition (the second *x*) is *bound* because its first occurrence in the definition is λ*x*.

- A variable that is not bound in `expr` is said to be *free* in `expr`. In the function (λ*x*.*xy*), the variable *x* in the body of the function is bound and the variable *y* is free.
- Every variable in a lambda expression is either bound or free. Bound and free variables have quite a different status in functions.
- In the expression (λ*x*.*x*)(λ*y*.*yx*):
  - The variable *x* in the body of the leftmost expression is bound to the first lambda.
  - The variable *y* in the body of the second expression is bound to the second lambda.
  - The variable *x* in the body of the second expression is free.
  - Note that *x* in second expression is independent of the *x* in the first expression.
- In the expression (λ*x*.*xy*)(λ*y*.*y*):
  - The variable *y* in the body of the leftmost expression is free.
  - The variable *y* in the body of the second expression is bound to the second lambda.
- Given an expression *e*, the following rules define FV(*e*), the set of free variables in *e*:
  - If *e* is a variable *x*, then FV(*e*) = {*x*}.
  - If *e* is of the form λ*x*.*y*, then FV(*e*) = FV(*y*) - {*x*}.
  - If *e* is of the form xy, then FV(*e*) = FV(*x*) ∪ FV(*y*).
- An expression with no free variables is said to be *closed*.

8. Beta Reductions

- A function application `λx.e f` is evaluated by substituting the argument `f` for the formal parameter `x` in the body `e` of the function definition.
- We will use the notation `[f/x]e` to indicate that `f` is to be substituted for all free occurrences of `x` in the expression `e`.
- Examples:

1. (λ*x*.*x*)*y* → [*y*/*x*]*x* = *y*.
2. (λ*x*.*xzx*)*y* → [*y*/*x*]*xzx* = *yzy*.
3. (λ*x*.*z*)*y* → [*y*/*x*]*z* = *z*.

- This substitution in a function application is called a *beta reduction* and we use a right arrow to indicate it.
- If expr1 → expr2, we say expr1 *reduces* to expr2 in one step.
- In general, `(λx.e)f → [f/x]e` means that applying the function `(λx.e)` to the argument expression `f` reduces to the expression `[f/x]e` where the argument expression `f` is substituted for the function's formal parameter `x` in the function body `e`.
- A lambda calculus expression (aka a "program") is "run" by computing a final result by repeatly applying beta reductions. We use →* to denote the reflexive and transitive closure of →; that is, zero or more applications of beta reductions.
- Examples:
  - (λ*x*.*x*)*y* → *y* (illustrating that λ*x*.*x* is the identity function).
  - (λ*x*.*xx*)(λ*y*.*y*) → (λ*y*.*y*)(λ*y*.*y*) → (λ*y*.*y*); thus, we can write (λ*x*.*xx*)(λ*y*.*y*) →* (λ*y*.*y*). Note that here we have applied a function to a function as an argument and the result is a function.

9. Evaluating a Lambda Expression

- A lambda calculus expression can be thought of as a program which can be executed by evaluating it. Evaluation is done by repeatedly finding a reducible expression (called a redex) and reducing it by a function evaluation until there are no more redexes.
- Example 1: The lambda expression (λ*x*.*x*)*y* in its entirety is a redex that reduces to *y*.
- Example 2: The lambda expression (+ (* 1 2) (- 4 3)) has two redexes: (* 1 2) and (- 4 3). If we choose to reduce the first redex, we get (+ 2 (- 4 3)). We can then reduce (+ 2 (- 4 3)) to get (+ 2 1). Finally we can reduce (+ 2 1) to get 3.
- Note that if we had chosen the second redex to revaluate first, we would have ended up with the same result:

  (+ (* 1 2) (- 4 3)) → (+ (* 1 2) 1) → (+ 2 1) → 3.

10. Currying

- All functions in lambda calculus are prefix and take exactly one argument.
- If we want to apply a function to more than one argument, we can use a technique called *currying* that treats a function applied to more than one argument to a sequence of applications of one-argument functions. For example, to express the sum of 1 and 2 we can write (+ 1 2) as ((+ 1) 2) where the expression (+ 1) denotes the function that adds 1 to its argument. Thus ((+ 1) 2) means the function + is applied to the argument 1 and the result is a function (+ 1) that adds 1 to its argument: (+ 1 2) = ((+ 1) 2) → 3
- Note that function application associates to the left.

11. Renaming Bound Variables by Alpha Reduction

- The name of a formal parameter in a function definition is arbitrary. We can use any variable to name a parameter, so that the function λ*x*.*x* is equivalent to λ*y*.*y* and λ*z*.*z*. This kind of renaming is called *alpha reduction*.
- Note that we cannot rename free variables in expressions.

- Also note that we cannot change the name of a bound variable in an expression to conflict with the name of a free variable in that expression.

12. Eta Conversion

- The two lambda expressions (λ*x*.+ 1 *x*) and (+ 1) are equivalent in the sense that these expressions behave in exactly the same way when they are applied to an argument -- they add 1 to it. η-conversion is a rule that expresses this equivalence.
- In general, if *x* does not occur free in the function *F*, then (λ*x*.*F x*) is η-convertible to *F*.

13. Substitutions

- For a beta reduction, we introduced the notation `[f/x]e` to indicate that the expression `f` is to be substituted for all free occurrences of the formal parameter `x` in the expression `e`:

  `(λx.e) f → [f/x]e`
- To avoid name clashes in a substitution `[f/x]e`, first rename the bound variables in `e` and `f` so they become distinct. Then perform the textual substituion of `f` for `x` in `e`.
  - For example, consider the substitution `[y(λx.x)/x] λy.(λx.x)yx`.
  - After renaming all the bound variables to make them all distinct we get `[y(λu.u)/x] λv.(λw.w)vx`.
  - Then doing the substitution we get `λv.(λw.w)v(y(λu.u))`.
- The rules for substitution are as follows. We assume `x` and `y` are distinct variables, and `e`, `f` and `g` are expressions.
  - For variables

    `[e/x]x = e`

    `[e/x]y = y`
  - For function applications

    `[e/x](f g) = ([e/x]f) ([e/x]g)`
  - For function abstractions

    `[e/x](λx.f) = λx.f`

    `[e/x](λy.f) = λy.[e/x]f`, provided `y` is not free in `e` (the "freshness" condition).
- Examples:

1. `[y/y](λx.x) = λx.x`
2. `[y/x]((λx.y) x) = ([y/x](λx.y)) ([y/x]x) = (λx.y)y`
3. Note that the freshness condition does not allow us to make the substitution `[y/x](λy.x) = λy.([y/x]x) = λy.y` because `y` is free in the expression `y`.

14. Disambiguating Lambda Expressions

- The grammar we gave in section 4 for lambda expressions is ambgious. A few simple rules will remove the ambiguities.
- Function application is left associative: *f g h* = ((*f g*) *h*)
- Function application binds more tightly than lambda: λ*x*.*f g x* = (λ*x*.(*f g*) *x*)
- The body in a function abstraction extends as far to the right as possible: λ*x*. + *x* 1 = λ*x*. (+ *x* 1).

15. Normal Form

- An expression containing no possible beta reductions is said to be in normal form. A normal form expression is one containing no redexes.
- Examples of normal form expressions:
  - `x` where `x` is a variable.
  - `x e` where `x` is a variable and `e` is a normal form expression.
  - `λx.e` where `x` is a variable and `e` is a normal form expression.
- The expression `(λx.x x)(λx.x x)` does not have a normal form because it always evaluates to itself. We can think of this expression as a representation for an infinite loop.

16. Evaluation Strategies

- An expression may contain more than one redex so there can be several reduction sequences. For example, the expression `(+ (* 1 2) (- 4 3))` can be reduced to normal form with two reduction sequences:

  `(+ (* 1 2) (- 4 3))`

  `→ (+ 2 (- 4 3))`

  `→ (+ 2 1)`

  `→ 3`

  or the sequence

  `(+ (* 1 2) (- 4 3))`

→ `(+ (* 1 2) 1)`

→ `(+ 2 1)`

→ `3`

- As we pointed out in (15), the expression `(λx.x x)(λx.x x)` does not have a terminating sequence of reductions.
- Some reduction sequences for a given expression may reach a normal form while others may not. For example, consider the expression `(λx.1)((λx.x x)(λx.x x))`. If we reduce the application of `(λx.1)` to `(λx.x x)(λx.x x)`, we get the result 1, but if we keep reducing the application of `(λx.x x)` to `(λx.x x)`, the evaluation will not terminate.
- We shall see that repeatedly reducing the leftmost outermost redex will always yield a normal form, if a normal form exists.

17. Practice Problems

1. Evaluate `(λx. λy. + x y)1 2`.
2. Evaluate `(λf. f 2)(λx. + x 1)`.
3. Evaluate `(λx. (λx. + (* 1)) x 2) 3`.
4. Evaluate `(λx. λy. + x((λx. * x 1) y))2 3`.
5. Is `(λx. + x x)` η-convertible to `(+ x)`?
6. Show how all bound variables in a lambda expression can be given distinct names.
7. Construct an unambiguous grammar for lambda calculus.

18. References

- Simon Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.
- Stephen A. Edwards: The Lambda Calculus
- http://www.inf.fu-berlin.de/lehre/WS01/ALPI/lambda.pdf
- http://www.soe.ucsc.edu/classes/cmps112/Spring03/readings/lambdacalculus/project3.html

---

aho@cs.columbia.edu

COMS W4115
Programming Languages and Translators
Lecture 23: April 22, 2013
The Lambda Calculus

Outline

- Reduction orders
- The Church-Rosser theorems
- The Y combinator
- Implementing factorial using the Y combinator
- Church numerals
- Arithmetic
- Logic
- Other programming language constructs
- The influence of the lambda calculus on functional languages

1. Reduction Orders

- Programming languages use many different techniques to pass parameters to procedures such as call by value, call by reference, call by value-return, call by name, and so on. We can model many of these parameter-passing mechanisms using the lambda calculus.
- The order in which reductions are applied within a lambda expression can affect the final result. We will use the terms reduction order and evaluation order synonymously.
- A reducible expression (redex) in a lambda expression is a subexpression that can be reduced using beta reduction.
- An expression that contains no redexes is said to be in normal form.
- Some reduction orders for an expression may yield a normal form expression while other orders may not. For example, consider the expression

  `(λx.1)((λx.x x)(λx.x x))`
- This expression has two redexes:

1. The entire expression is a redex in which we can apply the function `(λx.1)` to the argument `((λx.x x)(λx.x x))` to yield the value 1.
2. The rightmost subexpression `((λx.x x)(λx.x x))` is also a redex in which we can apply the function `(λx.x x)` to the argument `(λx.x x)`. But if we do this reduction we get same subexpression: `(λx.x x)(λx.x x)` → `(λx.x x)(λx.x x)`. Thus, continuing this order of evaluation will not terminate in a normal form.

- A remarkable property of lambda calculus is that every lambda expression has a unique normal form if one exists.
- The expression `(λx.1)((λx.x x)(λx.x x))` has the normal form 1.
- The expression `(λx.x x)(λx.x x)` does not have a normal form because it always evaluates to itself. We can think of this expression as a representation for an infinite loop.
- There are two common reduction orders for lambda expressions: normal order evaluation and applicative order evaluation.
- Normal order evaluation
  - In normal order evaluation we always reduce the leftmost outermost redex at each step.
  - The first reduction order above is a normal order evaluation.
  - If an expression has a normal form, then normal order evaluation will always find it.
- Applicative order evaluation
  - In applicative order evaluation we always reduce the leftmost innermost redex at each step.
  - The second reduction order above is an applicative order evaluation.
  - Thus, even though an expression may have a normal form, applicative order evaluation may fail to find it.

2. The Church-Rosser Theorems

- A remarkable property of lambda calculus is that every expression has a unique normal form if one exists.
- **Church-Rosser Theorem I:** If `e` $\rightarrow^*$ `f` and `e` $\rightarrow^*$ `g` by any two reduction orders, then there always exists a lambda expression `h` such that `f` $\rightarrow^*$ `h` and `g` $\rightarrow^*$ `h`.
  - A corollary of this theorem is that no lambda expression can be reduced to two distinct normal forms. To see this, suppose `f` and `g` are in normal form. The Church-Rosser theorem says there must be an expression `h` such that `f` and `g` are each reducible to `h`. Since `f` and `g` are in normal form, they cannot have any redexes so `f = g = h`.
  - This corollary says that all reduction sequences that terminate will always yield the same result and that result must be a normal form.
  - The term *confluent* is often applied to a rewriting system that has the Church-Rosser property.
- **Church-Rosser Theorem II:** If `e` $\rightarrow^*$ `f` and `f` is in normal form, then there exists a normal order reduction sequence from `e` to `f`.

3. The Y Combinator

- The `Y` combinator (sometimes called the paradoxical combinator) is a function that takes a function `G` as an argument and returns `G(YG)`. With repeated applications we can get `G(G(YG)), G(G(G(YG))),...` .
- We can implement recursive functions using the `Y` combinator.
- `Y` is defined as follows:

  ```
  (λf.(λx.f(x x))(λx.f(x x)))
  ```

- Let us evaluate `YG` where `G` is any expression:

  ```
  (λf.(λx.f(x x))(λx'.f(x' x'))) G
  → (λx.G(x x))(λx'.G(x' x'))
  → G((λx'.G(x' x'))(λx'.G(x' x')))
  ↔ G((λf.(λx.f(x x))(λx.f(x x)))G)
  = G(YG)
  ```

- Thus, `YG` $\rightarrow^*$ `G(YG)`; that is, `YG` reduces to a call of `G` on `(YG)`.
- We will use `Y` to implement recursive functions.
- `Y` is an example of a fixed-point combinator.

4. Implementing Factorial using the Y Combinator

- If we could name lambda abstractions, we could define the factorial function with the following recursive definition:

  ```
  FAC = (λn.IF (= n 0) 1 (* n (FAC (- n 1 ))))
  ```

  where `IF` is a conditional function.
- However, functions in lambda calculus cannot be named; they are anonymous.
- But we can express recursion as the fixed-point of a function `G`. To do this, let us simplify the essence of the problem. We begin with a skeletal recursive definition:

  ```
  FAC = λn.(... FAC ...)
  ```

- By performing beta abstraction on `FAC`, we can transform its definition to:

  ```
  FAC = (λf.(λn.(... f ...))) FAC
     = G FAC
  ```

where

```
G = λf.λn.IF (= n 0) 1 (* n (f (- n 1 )))
```

Beta abstraction is just the reverse of beta reduction.

- The equation

  ```
  FAC = G FAC
  ```

  says that when the function G is applied to FAC, the result is FAC. That is, FAC is a fixed-point of G.
- We can use the Y combinator to implement FAC:

  ```
  FAC = Y G
  ```
- As an example, let compute `FAC 1`:

```
FAC 1 = Y G 1
      = G (Y G) 1
      = λf.λn.IF (= n 0) 1 (* n (f (- n 1 ))))(Y G) 1
      → λn.IF (= n 0) 1 (* n ((Y G) (- n 1 ))))1
      → IF (= n 0) 1 (* n ((Y G) (- 1 1 )))
      → * 1 (Y G 0)
      = * 1 (G(Y G) 0)
      = * 1((λf.λn.IF (= n 0) 1 (* n (f (- n 1 ))))(Y G 0)
      → * 1((λn.IF (= n 0) 1 (* n ((Y G) (- n 1 ))))0
      → * 1(IF (= 0 0) 1 (* 0 ((Y G) (- 0 1 )))
      → * 1 1
      → 1
```

5. Church Numerals

- Church numberals are a way of representing the integers in lambda calculus.
- Church numerals are defined as functions taking two parameters:

  `0` is defined as `λf.λx. x`

  `1` is defined as `λf.λx. f x`

  `2` is defined as `λf.λx. f (f x)`.

  `3` is defined as `λf.λx. f (f (f x))`.

  n is defined as $\lambda f.\lambda x.\ f^n\ x$
- n has the property that for any lambda expressions g and y, ngy $\rightarrow^*$ $g^n y$. That is to say, ngy causes g to be applied to y *n* times.

6. Arithmetic

- In lambda calculus, arithmetic functions can be represented by corresponding operations on Church numerals.
- We can define a successor function `succ` of three arguments that adds one to its first argument:

  ```
  λn.λf.λx. f (n f x)
  ```
  - Example: Let us evaluate `succ 0`:

  ```
  (λn.λf.λx. f (n f x)) 0
    → λf.λx. f (0 f x)
    = λf.λx. f ((λf.λx. x) f x)
    → λf.λx. f (λx. x) x)
    → λf.λx. f x
    = 1
  ```
- We can define a function `add` using the identity $f^{(m+n)} = f^m \circ f^n$ as follows:

  ```
  λm.λn.λf.λx. m f (n f x)
  ```
  - Example: Let us evaluate `add 1 2`:

  ```
  λm.λn.λf.λx. m f (n f x) 1 2
    →  λn.λf.λx. 1 f (n f x) 2
    →* λf.λx. f (f (f x))
    =  3
  ```

7. Logic

- The boolean value true can be represented by a function of two arguments that always selects its first argument: `λx.λy.x`
- The boolean value false can be represented by a function of two arguments that always selects its second argument: `λx.λy.y`
- An if-then-else statement can be represented by a function of three arguments `λc.λi.λe. c i e` that uses its condition c to select either the if-part `i` or the else-part `e`.
  - Example: Let us evaluate if true then 1 else 2:

```
(λc.λi.λe. c i e) true 1 2
  → (λi.λe. true i e) 1 2
  → (λe. true 1 e) 2
  → true 1 2
  = (λx.λy.x) 1 2
  → (λy.1) 2
  → 1
```

- The boolean operators and, or, and not can be implemented as follows:

```
and = λp.λq. p q p
or  = λp.λq. p p q
not = λp.λa.λb. p b a
```

  - Example: Let us evaluate `not true`:

```
(λp.λa.λb. p b a) true
  → λa.λb. true b a
  = λa.λb. (λx.λy.x) b a
  → λa.λb. (λy.b) a
  → λa.λb. b
  = false (under renaming)
```

8. Other Programming Language Constructs

- We can readily implement other programming language constructs in lambda calculus. As an example, here are lambda calculus expressions for various list operations such as cons (constructing a list), head (selecting the first item from a list), and tail (selecting the remainder of a list after the first item):

```
cons = λh.λt.λf. f h t
head = λl.l (λh.λt. h)
tail = λl.l (λh.λt. t)
```

9. The Influence of The Lambda Calculus on Functional Languages

- Our next lecture will be by Maria Taku who will talk about the influence of the lambda calculus on functional languages and her experiences implementing the PLT compiler project using OCaml.

10. Practice Problems

1. Evaluate `(λx.((λw.λz. + w z)1)) ((λx. xx)(λx. xx))) ((λy. * y 1) (- 3 2))` using normal order evaluation and applicative order evaluation.
2. Give an example of a code optimization transformation that has the Church-Rosser property.
3. Evaluate `FAC 2`.
4. Evaluate `succ two`.
5. Evaluate `add two three`.
6. Let `mul` be the function

   ```
   λm.λn.λf.λx. m (n f x)
   ```

   Evaluate `mul two three`.

- Write a lambda expression for the boolean predicate `isZero` and evaluate `isZero one`.

11. References

- Simon Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.
- Stephen A. Edwards: The Lambda Calculus
- http://www.inf.fu-berlin.de/lehre/WS01/ALPI/lambda.pdf
- http://www.soe.ucsc.edu/classes/cmps112/Spring03/readings/lambdacalculus/project3.html

---

aho@cs.columbia.edu