CS 314 Principles of Programming Languages

Lecture 1: Overview and Basics

Prof. Zheng Zhang

Rutgers University

September 5, 2018

Why study the ***principles*** of programming languages?

Why study the ***principles*** of programming languages?

1. **Better Understanding of Programming Language (PL) Theory**

- Programming language defines computation models tailored to *think about* and *solve* problems
- It is believed that the depth at which we think is influenced by the expressive power of the language which we communicate our thoughts
- While it may be possible to solve every problem in any reasonable language, some problems are inherently suitable for specific ways of thinking and programming.

Why study the ***principles*** of programming languages?

1. **Better Understanding of Programming Language (PL) Theory**

- Programming language defines computation models tailored to *think about* and *solve* problems
- It is believed that the depth at which we think is influenced by the expressive power of the language which we communicate our thoughts
- While it may be possible to solve every problem in any reasonable language, some problems are inherently suitable for specific ways of thinking and programming.

In 2009, Twitter switched (part of its server infrastructure) from Ruby on Rails to Scala because Scala better matched their need for long running threads, high performance under heavy loads, more robust code via compile-time type checking, and ease of composable functions rooted in functional programming.

— "Twitter on Scala"

A Conversation with Steve Jenson, Alex Payne, and Robey Pointer

Why study the ***principles*** of programming languages?

1. **Better Understanding of Programming Language (PL) Theory**

- Programming language defines computation models tailored to *think about* and *solve* problems
- It is believed that the depth at which we think is influenced by the expressive power of the language which we communicate our thoughts
- While it may be possible to solve every problem in any reasonable language, some problems are inherently suitable for specific ways of thinking and programming.

Why study the ***principles*** of programming languages?

1. **Better Understanding of Programming Language (PL) Theory**

- Programming language defines computation models tailored to *think about* and *solve* problems
- It is believed that the depth at which we think is influenced by the expressive power of the language which we communicate our thoughts
- While it may be possible to solve every problem in any reasonable language, some problems are inherently suitable for specific ways of thinking and programming.

> *Using LISP for my company's store front application, which eventually became "Yahoo Store", allow us to develop and deploy new functionality more rapidly than our several dozen competitors (who primarily use C++ and CGI script). Elements of LISP, meta-programming enable programs to create, modify, and execute new pieces of code, and make implementing complex features much easier.*
>
> — Paul Graham.
>
> Hackers and Painters: Big Ideas from the Computer Age.

Why study the ***principles*** of programming languages?

1. Better **Understanding of Programming Language (PL) Theory**

- Programming language defines computation models tailored to *think about* and *solve* problems.
- It is believed that the depth at which we think is influenced by the expressive power of the language which we communicate our thoughts.
- While it may be possible to solve every problem in any reasonable language, some problems are inherently suitable for specific ways of thinking and programming.

Why study the ***principles*** of programming languages?

1. Better **Understanding of Programming Language (PL) Theory**

- Programming language defines computation models tailored to *think about* and *solve* problems.
- It is believed that the depth at which we think is influenced by the expressive power of the language which we communicate our thoughts.
- While it may be possible to solve every problem in any reasonable language, some problems are inherently suitable for specific ways of thinking and programming.

Understanding the fundamentals in programming languages helps one critically *compare* and *choose* the most appropriate abstractions for describing and solving a particular problem.

Why study the ***principles*** of programming languages?

2. Better **Learning of New Languages**

- The languages and models in practice change constantly.
- Advances in our fields change how we model and express computation.

Why study the ***principles*** of programming languages?

2. Better **Learning of New Languages**

- The languages and models in practice change constantly.
- Advances in our fields change how we model and express computation.

  Many widely used languages such as Java, C#, go, and many scripting languages, largely perform automatic memory management via *garbage collection* because of two things:
  - Advance in processor and memory performance
  - Improvement in collection techniques

Why study the ***principles*** of programming languages?

2. Better **Learning of New Languages**

- The languages and models in practice change constantly.
- Advances in our fields change how we model and express computation.

  Many widely used languages such as Java, C#, go, and many scripting languages, largely perform automatic memory management via *garbage collection* because of two things:
  - Advance in processor and memory performance
  - Improvement in collection techniques

Garbage collection is a technique that was invented by John McCarthy around 1959 to simplify manual memory management in Lisp.

The Very Basics of Garbage Collection
http://www.memorymanagement.org

Why study the ***principles*** of programming languages?

2. Better **Learning of New Languages**

- The languages and models in practice change constantly.
- Advances in our fields change how we model and express computation.

Why study the ***principles*** of programming languages?

2. Better **Learning of New Languages**

- The languages and models in practice change constantly.
- Advances in our fields change how we model and express computation.
- Need to understand fundamental principles underlying existing PLs, and have prior experience with a variety of computational models.

Why study the ***principles*** of programming languages?

2. Better **Learning of New Languages**

- The languages and models in practice change constantly.
- Advances in our fields change how we model and express computation.
- Need to understand fundamental principles underlying existing PLs, and have prior experience with a variety of computational models.
- The knowledge will endure longer than today's "hot" languages, and let programmers quickly look beyond an unfamiliar language's surface-level details, grasp the underlying computational model's design principles.

Why study the ***principles*** of programming languages?

3. Better **Design of Domain Specific Languages (DSL)**

- One may not have to design general purpose programming language, but very likely domain-specific API, programming language, and virtual machine (VM) during his/her career.
- DSL exploits properties of the intended domains to facilitate writing specific types of algorithms;

Why study the ***principles*** of programming languages?

3. Better **Design of Domain Specific Languages (DSL)**

- One may not have to design general purpose programming language, but very likely domain-specific API, programming language, and virtual machine (VM) during his/her career.
- DSL exploits properties of the intended domains to facilitate writing specific types of algorithms;

  Examples

  - MATLAB and Mathematica for manipulating math form
  - Verilog and VHDL for hardware circuits
  - Latex for typesetting documents
  - Cg for rendering algorithms on graphics

Why study the ***principles*** of programming languages?

3. Better **Design of Domain Specific Languages (DSL)**

- One may not have to design general purpose programming language, but very likely domain-specific API, programming language, and virtual machine (VM) during his/her career.
- DSL exploits properties of the intended domains to facilitate writing specific types of algorithms;

Why study the ***principles*** of programming languages?

3. Better **Design of Domain Specific Languages (DSL)**

- One may not have to design general purpose programming language, but very likely domain-specific API, programming language, and virtual machine (VM) during his/her career.
- DSL exploits properties of the intended domains to facilitate writing specific types of algorithms;
- However, lack of knowledge of programming fundamentals can lead to DSL that are difficult to understand/use or need later repair.

Example: *in earlier versions of LISP, the dynamic scope make higher order abstractions unusable in many cases and lead to problems in type checking and optimizations. This problem had to be fixed in LISP.*

Course Goals

- To gain understanding of the basic structure of programming languages:
  - Data visibility, naming conventions, procedure abstraction and etc.
- To study different computing models and language paradigms:
  - Functional (LISP/*Scheme*), imperative (*C*), logic (*Prolog*), object-oriented (Java/*C++*), parallel (*OpenMP*/*CUDA*), …
  - To ensure an appropriate language is chosen for a task
- To know the principles underlying all programming languages:
  - To make learning new programming languages easier
  - To enable full use of a programming language
  - To understand the implementation of different programming languages

Programming languages embody many concepts central to all of computer science, including abstraction, generalization and automation, computability, and resource management.

Course Information

Prerequisites:

- CS 205 (Introduction to Discrete Structures)
- CS 211 (Computer Architecture)

Important facts:

- Lecturer: Prof. Zheng Zhang
- Lectures: Wednesday 10:20am - 11:40am
  Friday 3:20pm - 4:40pm
- TAs: Yanhao Chen
  Junchi Jiang
  Lau Chun Leung
- Recitations: TA/Recitation Assignment TBA
- Office Hours: TBA

Recitations and Office Hours Start Next Week

Course Information

Basis for grades (subject to changes):

- 10% homework (about 8 homework sets, lowest one will be dropped)
- 25% mid-term exam
- 35% final exam (cumulative)
- 30% three major programming projects
- Up to 5% extra-credit work in homework or projects

If you do better in final exam than in mid-term, your midterm grade will be dropped, and your final exam will count 60% of the final grade.

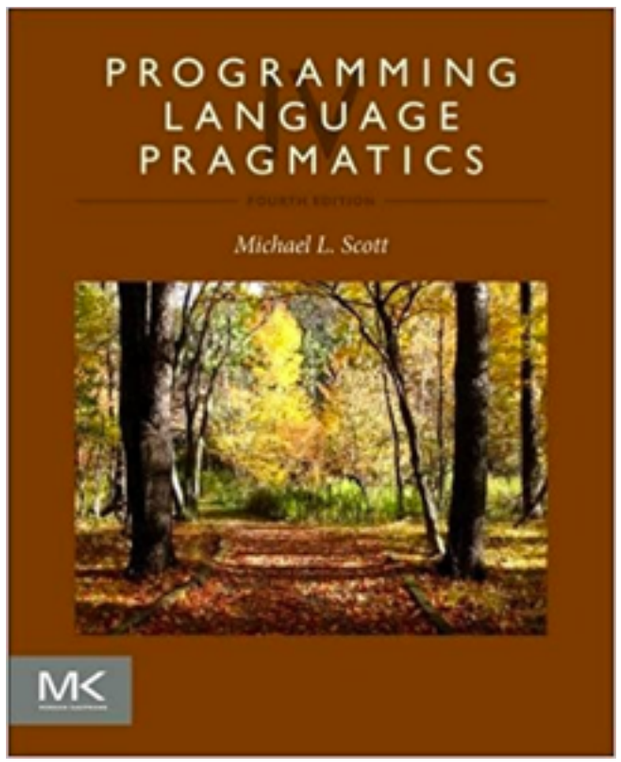
Course Information

Textbook:

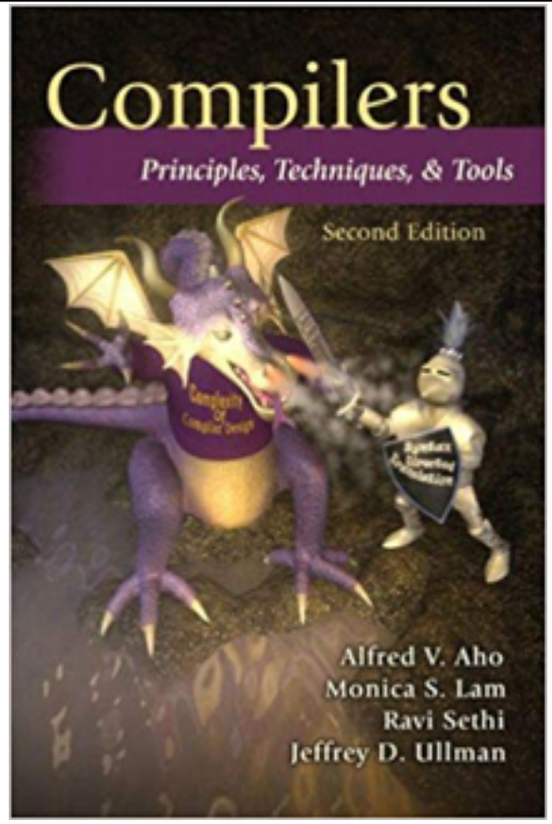
- Required:
  "Programming Language Pragmatics" by Michael L. Scott, 4th Edition, Morgan Kaufmann (Elsevier), 2015.

- Suggested:
  "Compilers: Principles, Techniques, and Tools 2/E (2007) " by Aho, Lam, Sethi, Ullman, 2nd Edition, Addison-Wesley, 2007

Required: Scott



Suggested: ALSU



Course Information

Online discussion:

- There will be a discussion forum on Sakai.
  - All questions regarding homework and projects should be posted here.
  - Please DO NOT post solutions.
- All questions will be answered in Sakai (in no more than 72 hours).
  - Before you ask a question, check if a similar question has been asked.
  - Plan ahead and try not to ask questions in the last minute.
  - Be specific and avoid asking general questions.

Course Information

Academic Integrity

- Read-protect your directories and files (ilab)
- No group projects
- Will use software tools to check code plagiarism

IMPORTANT INFORMATION will be posted on Sakai

- Grading of homework and projects
- Instructions of how to submit programming projects

Email TAs and me:

- Subject line has to start with 314
  E.g., “314: Question about my midterm exam”
- No project and homework questions via email;
  Please post them on Sakai discussion forum.

Course Information

Special permission numbers:

- Currently all three sections are full.
- Will try to accommodate as many students as I can.
- Talk to me after class.

Why Use Anything Besides Machine Code?

This is a **C** program that uses two one-dimensional arrays **a** and **b** of size **SIZE**. The arrays are initialized, and then a sum operation is performed. The size of the arrays and the result of the sum is printed out.

```
#include <stdio.h>
#define SIZE 100
int main() {
  int a[SIZE], b[SIZE];
  int i, sum;
  for (i=0; i<SIZE; i++) {
    a[i] = 1;
    b[i] = 2;
  }
  sum = 0;
  for (i=0; i<SIZE; i++)
    sum = sum + a[i] + b[i];
  printf("for two arrays of size %d, sum = %d\n", SIZE, sum);
}
```

14 Lines in Total

Why Use Anything Besides Machine Code?

Compiler: gcc -O3 -S example.c —> example.s

```
.file "example.c"
.version "01.01"
gcc2_compiled.:
.section .rodata.str1.32,"aMS",@progbits,1
.align 32
.LC0:
.string "for two arrays of size %d, sum = %d\n" .text
.align 4
.globl main
.type main,@function
main:
pushl %ebp
movl %esp, %ebp
xorl %eax, %eax
subl $808, %esp
movl $99, %edx
.p2align 2
.L21:
movl $1, -408(%ebp,%eax)
movl $2, -808(%ebp,%eax)
addl $4, %eax
decl %edx
jns .L21
xorl %ecx, %ecx
xorl %eax, %eax
movl $99, %edx
.p2align 2
```

```
.L26:
addl -408(%ebp,%eax), %ecx
addl -808(%ebp,%eax), %ecx
addl $4, %eax
decl %edx
jns .L26
pushl %eax
pushl %ecx
pushl $100
pushl $.LC0
call printf
addl $16, %esp
leave
ret
.Lfe1:
.size  main,.Lfe1-main
.ident "GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3 2.96-112)"
```

45 Lines in Total

Why Use Anything Besides Machine Code?

gcc -o example.o -O3 example.c;
strip example.o; objdump -d example.o;

```
objdump: example.o: No symbols

example.o:     file format elf32-sparc

Disassembly of section .text:

00010444 <.text>:
   10444: bc 10 20 00   clr  %fp
   10448: e0 03 a0 40   ld  [ %sp + 0x40 ], %l0
   1044c: a2 03 a0 44   add  %sp, 0x44, %l1
   10450: 9c 23 a0 20   sub  %sp, 0x20, %sp
   10454: 80 90 00 01   tst  %g1
   10458: 02 80 00 04   be  0x10468
   1045c: 90 10 00 01   mov  %g1, %o0
   10460: 40 00 40 c4   call  0x20770
   10464: 01 00 00 00   nop
   10468: 11 00 00 41   sethi  %hi(0x10400), %o0
   1046c: 90 12 22 d8   or  %o0, 0x2d8, %o0  ! 0x106d8
   10470: 40 00 40 c0   call  0x20770
   10474: 01 00 00 00   nop
   10478: 40 00 00 91   call  0x106bc
   1047c: 01 00 00 00   nop
   10480: 90 10 00 10   mov  %l0, %o0
   10484: 92 10 00 11   mov  %l1, %o1
   10488: 95 2c 20 02   sll  %l0, 2, %o2
   1048c: 94 02 a0 04   add  %o2, 4, %o2
   10490: 94 04 40 0a   add  %l1, %o2, %o2
   10494: 17 00 00 82   sethi  %hi(0x20800), %o3
```

```
10498:96 12 e0 a8   or  %o3, 0xa8, %o3     ! 0x208a8
1049c:d4 22 c0 00   st  %o2, [ %o3 ]
104a0:40 00 00 4e   call  0x105d8
104a4:01 00 00 00   nop
104a8:40 00 40 b5   call  0x2077c
104ac:01 00 00 00   nop
104b0:40 00 40 b6   call  0x20788
104b4:01 00 00 00   nop
104b8:81 c3 e0 08   retl
104bc:ae 03 c0 17   add  %o7, %l7, %l7
104c0:9d e3 bf 90   save  %sp, -112, %sp
104c4:11 00 00 00   sethi  %hi(0), %o0
104c8:2f 00 00 40   sethi  %hi(0x10000), %l7
104cc:7f ff ff fb call  0x104b8
104d0:ae 05 e2 54   add  %l7, 0x254, %l7  ! 0x10254
104d4:90 12 20 0c   or  %o0, 0xc, %o0
104d8:d2 05 c0 08   ld  [ %l7 + %o0 ], %o1
104dc:d4 02 40 00   ld  [ %o1 ], %o2
104e0:80 a2 a0 00   cmp  %o2, 0
104e4:12 80 00 23   bne  0x10570
104e8:11 00 00 00   sethi  %hi(0), %o0
104ec:90 12 20 10   or  %o0, 0x10, %o0     ! 0x10
104f0: d4 05 c0 08  ld  [ %l7 + %o0 ], %o2
104f4: d2 02 80 00  ld  [ %o2 ], %o1
104f8: d0 02 40 00  ld  [ %o1 ], %o0
104fc: 80 a2 20 00  cmp  %o0, 0
10500:02 80 00 0f   be  0x1053c
10504:11 00 00 00   sethi  %hi(0), %o0
10508:a0 10 00 0a   mov  %o2, %l0
1050c:d0 04 00 00       ld  [ %l0 ], %o0
10510:90 02 20 04 add  %o0, 4, %o0
10514:d0 24 00 00       st  %o0, [ %l0 ]
```

```
10518: d2 02 3f fc    ld  [ %o0 + -4 ], %o1
1051c: 9f c2 40 00    call  %o1
10520: 01 00 00 00    nop
10524: d0 04 00 00    ld  [ %l0 ], %o0
10528: d2 02 00 00    ld  [ %o0 ], %o1
1052c: 80 a2 60 00    cmp  %o1, 0
10530: 12 bf ff f9    bne  0x10514
10534: 90 02 20 04    add  %o0, 4, %o0
10538: 11 00 00 00    sethi  %hi(0), %o0
1053c: 90 12 20 1c    or  %o0, 0x1c, %o0     ! 0x1c
10540: d2 05 c0 08    ld  [ %l7 + %o0 ], %o1
10544: 80 a2 60 00    cmp  %o1, 0
10548: 02 80 00 05    be  0x1055c
1054c: 13 00 00 00    sethi  %hi(0), %o1
10550: 92 12 60 08    or  %o1, 8, %o1    ! 0x8
10554: 40 00 40 90    call  0x20794
10558: d0 05 c0 09    ld  [ %l7 + %o1 ], %o0
1055c: 11 00 00 00    sethi  %hi(0), %o0
10560: 90 12 20 0c    or  %o0, 0xc, %o0       ! 0xc
10564: d4 05 c0 08    ld  [ %l7 + %o0 ], %o2
10568: 92 10 20 01    mov  1, %o1
1056c: d2 22 80 00    st  %o1, [ %o2 ]
10570: 81 c7 e0 08    ret
10574: 81 e8 00 00    restore
10578: 9d e3 bf 90    save  %sp, -112, %sp
1057c: 81 c7 e0 08    ret
10580: 81 e8 00 00    restore
10584: 9d e3 bf 90    save  %sp, -112, %sp
10588: 11 00 00 00    sethi  %hi(0), %o0
1058c: 2f 00 00 40    sethi  %hi(0x10000), %l7
10590: 7f ff ff ca        call  0x104b8
10594: ae 05 e1 90    add  %l7, 0x190, %l7  ! 0x10190
10598: 90 12 20 18    or  %o0, 0x18, %o0
```

```
1059c: d2 05 c0 08   ld  [ %l7 + %o0 ], %o1
105a0: 80 a2 60 00   cmp  %o1, 0
105a4: 02 80 00 08   be  0x105c4
105a8: 13 00 00 00   sethi  %hi(0), %o1
105ac: 92 12 60 08   or  %o1, 8, %o1   ! 0x8
105b0: 15 00 00 00   sethi  %hi(0), %o2
105b4: d0 05 c0 09   ld  [ %l7 + %o1 ], %o0
105b8: 94 12 a0 04   or  %o2, 4, %o2
105bc: 40 00 40 79   call  0x207a0
105c0: d2 05 c0 0a   ld  [ %l7 + %o2 ], %o1
105c4: 81 c7 e0 08   ret
105c8: 81 e8 00 00   restore
105cc: 9d e3 bf 90   save  %sp, -112, %sp
105d0: 81 c7 e0 08   ret
105d4: 81 e8 00 00   restore
105d8: 9d e3 bc 70   save  %sp, -912, %sp
105dc: 92 07 be 60   add  %fp, -416, %o1
105e0: 94 07 bc d0   add  %fp, -816, %o2
105e4: 86 10 00 09   mov  %o1, %g3
105e8: 84 10 00 0a   mov  %o2, %g2
105ec: 9a 10 20 01   mov  1, %o5
105f0:  98 10 20 02   mov  2, %o4
105f4:  90 10 20 00   clr  %o0
105f8:  96 10 20 63   mov  0x63, %o3
105fc:  da 22 00 03   st  %o5, [ %o0 + %g3 ]
10600: d8 22 00 02   st  %o4, [ %o0 + %g2 ]
10604: 96 82 ff ff       addcc  %o3, -1, %o3
10608: 1c bf ff fd       bpos  0x105fc
1060c: 90 02 20 04   add  %o0, 4, %o0
10610: 9a 10 00 0a   mov  %o2, %o5
10614: 84 10 00 09   mov  %o1, %g2
10618: 94 10 20 00   clr  %o2
1061c: 98 10 20 00   clr  %o4
10620: 96 10 20 63   mov  0x63, %o3
```

```
10624: d0 03 00 02  ld  [ %o4 + %g2 ], %o0
10628: 96 82 ff ff  addcc  %o3, -1, %o3
1062c: d2 03 00 0d  ld  [ %o4 + %o5 ], %o1
10630: 90 02 80 08  add  %o2, %o0, %o0
10634: 94 02 00 09  add  %o0, %o1, %o2
10638: 1c bf ff fb  bpos  0x10624
1063c: 98 03 20 04  add  %o4, 4, %o4
10640: 11 00 00 41  sethi  %hi(0x10400), %o0
10644: 90 12 22 f8  or  %o0, 0x2f8, %o0    ! 0x106f8
10648: 40 00 40 59  call  0x207ac
1064c: 92 10 20 64  mov  0x64, %o1
10650: 81 c7 e0 08  ret
10654: 81 e8 00 00  restore
10658: 81 c3 e0 08  retl
1065c: ae 03 c0 17  add  %o7, %l7, %l7
10660: 9d e3 bf 90  save  %sp, -112, %sp
10664: 11 00 00 00  sethi  %hi(0), %o0
10668: 2f 00 00 40  sethi  %hi(0x10000), %l7
1066c: 7f ff ff fb  call  0x10658
10670: ae 05 e0 b4  add  %l7, 0xb4, %l7    ! 0x100b4
10674: 90 12 20 14  or  %o0, 0x14, %o0
10678: d2 05 c0 08  ld  [ %l7 + %o0 ], %o1
1067c: d4 02 7f fc  ld  [ %o1 + -4 ], %o2
10680: 80 a2 bf ff  cmp  %o2, -1
10684: 02 80 00 09  be  0x106a8
10688: a0 02 7f fc  add  %o1, -4, %l0
1068c: d0 04 00 00  ld  [ %l0 ], %o0
10690: 9f c2 00 00  call  %o0
10694: a0 04 3f fc  add  %l0, -4, %l0
10698: d0 04 00 00  ld  [ %l0 ], %o0
1069c: 80 a2 3f ff  cmp  %o0, -1
106a0: 12 bf ff fb  bne  0x1068c
```

```
   106a4:01 00 00 00   nop
   106a8:81 c7 e0 08   ret
   106ac:81 e8 00 00   restore
   106b0:9d e3 bf 90   save  %sp, -112, %sp
   106b4:81 c7 e0 08   ret
   106b8:81 e8 00 00   restore
Disassembly of section .init:

000106bc <.init>:
   106bc:9d e3 bf a0   save  %sp, -96, %sp
   106c0:7f ff ff b1       call  0x10584
   106c4:01 00 00 00   nop
   106c8:7f ff ff e6       call  0x10660
   106cc:01 00 00 00   nop
   106d0:81 c7 e0 08   ret
   106d4:81 e8 00 00   restore
Disassembly of section .fini:

000106d8 <.fini>:
   106d8:9d e3 bf a0   save  %sp, -96, %sp
   106dc:7f ff ff 79       call  0x104c0
   106e0:01 00 00 00   nop
   106e4:81 c7 e0 08   ret
   106e8:81 e8 00 00   restore
Disassembly of section .plt:

00020740 <.plt>:
       ...
   20770:03 00 00 30   sethi  %hi(0xc000), %g1
   20774:30 bf ff f3       b,a   0x20740
   20778:01 00 00 00   nop
   2077c:03 00 00 3c   sethi  %hi(0xf000), %g1
   20780:30 bf ff f0       b,a   0x20740
   20784:01 00 00 00   nop
```

```
20788:03 00 00 48   sethi  %hi(0x12000), %g1
2078c:30 bf ff ed   b,a   0x20740
20790:01 00 00 00   nop
20794:03 00 00 54   sethi  %hi(0x15000), %g1
20798:30 bf ff ea   b,a   0x20740
2079c:01 00 00 00   nop
207a0:03 00 00 60   sethi  %hi(0x18000), %g1
207a4:30 bf ff e7   b,a   0x20740
207a8:01 00 00 00   nop
207ac:03 00 00 6c   sethi  %hi(0x1b000), %g1
207b0:30 bf ff e4   b,a   0x20740
207b4:01 00 00 00   nop
207b8:01 00 00 00   nop
```

Results of applying:

gcc -o example.o -O3 example.c;

strip example.o;objdump -d example.o;

207 Lines in Total

Why Use Anything Besides Machine Code?

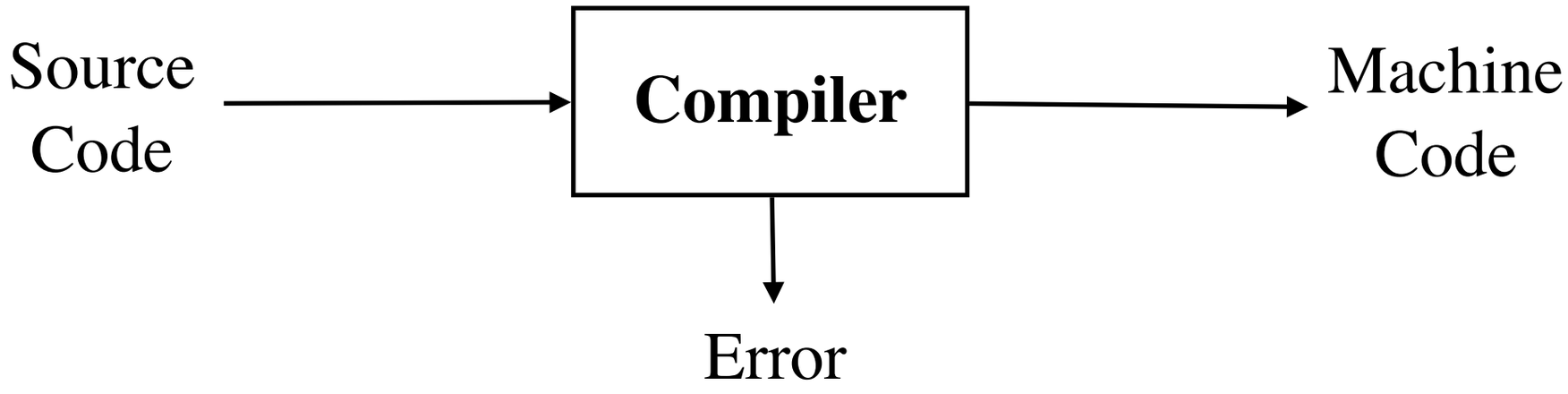
Need for high-level programming languages for

- Readable, familiar notations
- Machine independence (portability)
- Consistency checks during implementation
- Dealing with scale

The art of programming is the art of organizing complexity.

- Dijkstra, 1972

Compilers

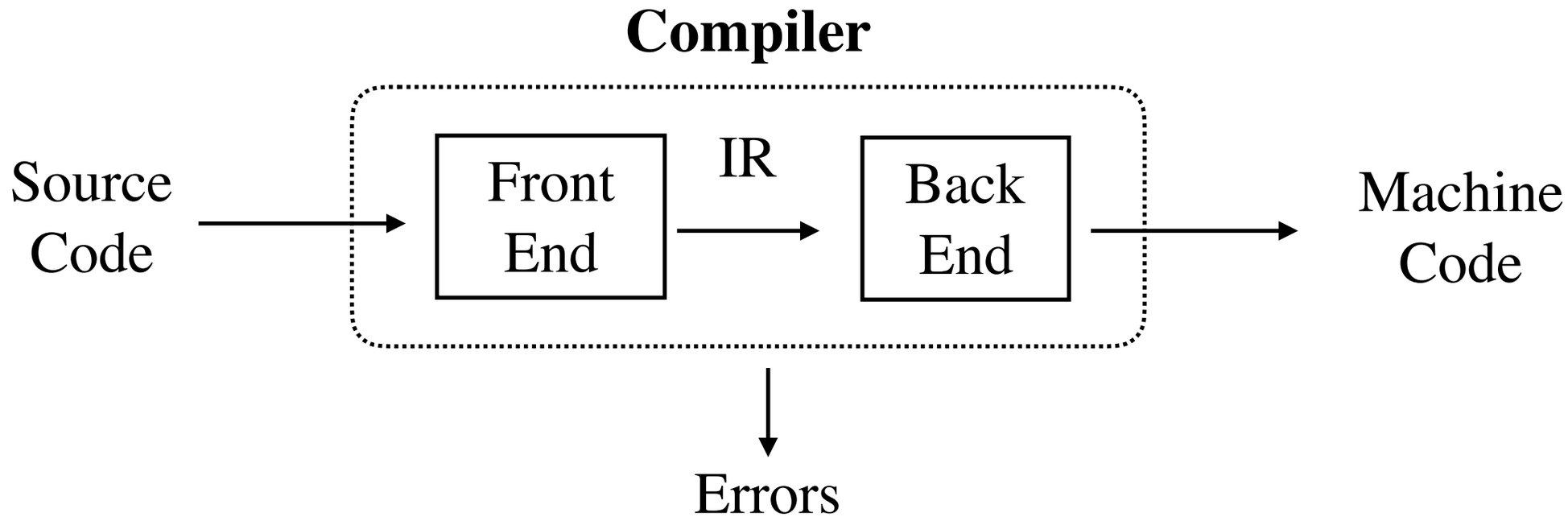


Implications:

- Recognize legal (and illegal) programs
- Generate correct code
- Manage storage of all variables and code
- Need format for object (or assembly) code

Big step up from assembler to higher level notations

Traditional Two-Pass Compilers



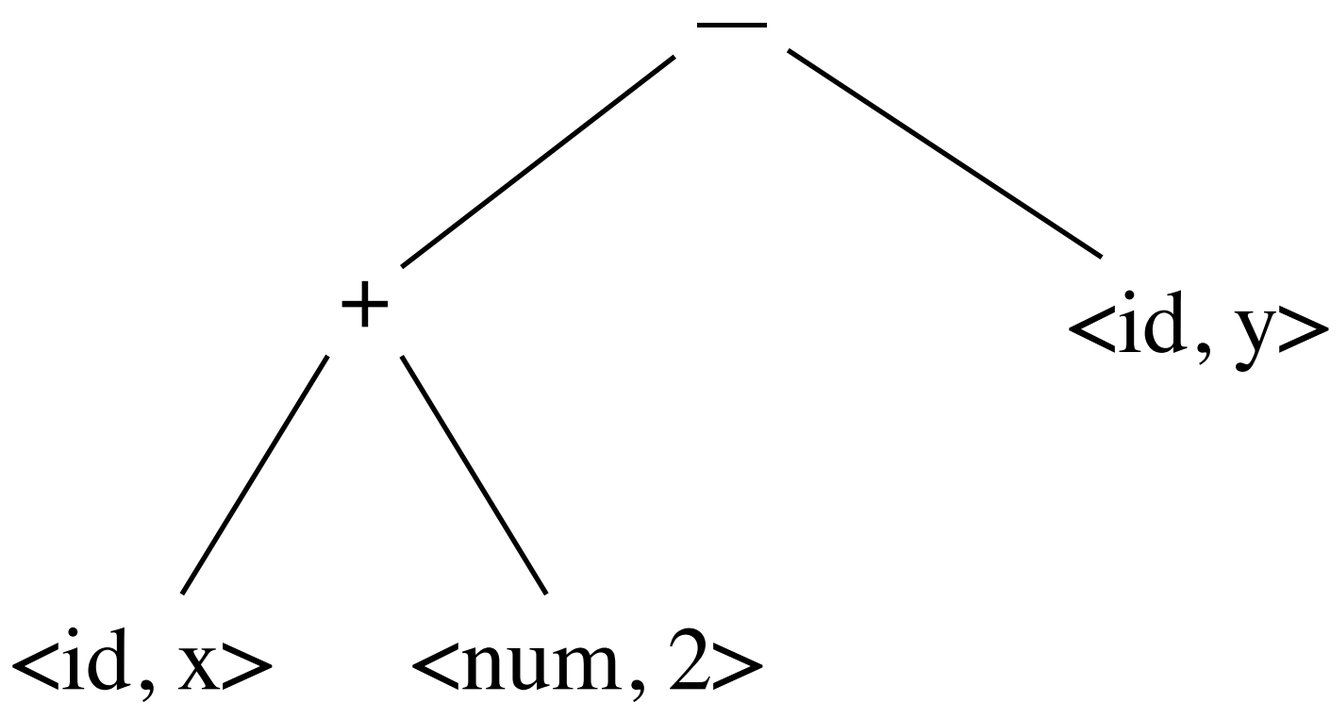
Implications:

- Intermediate representation (IR)
- Front end maps legal code into IR
- Back end maps IR onto target machine
- Simplify retargeting
- Allows multiple front ends and back ends

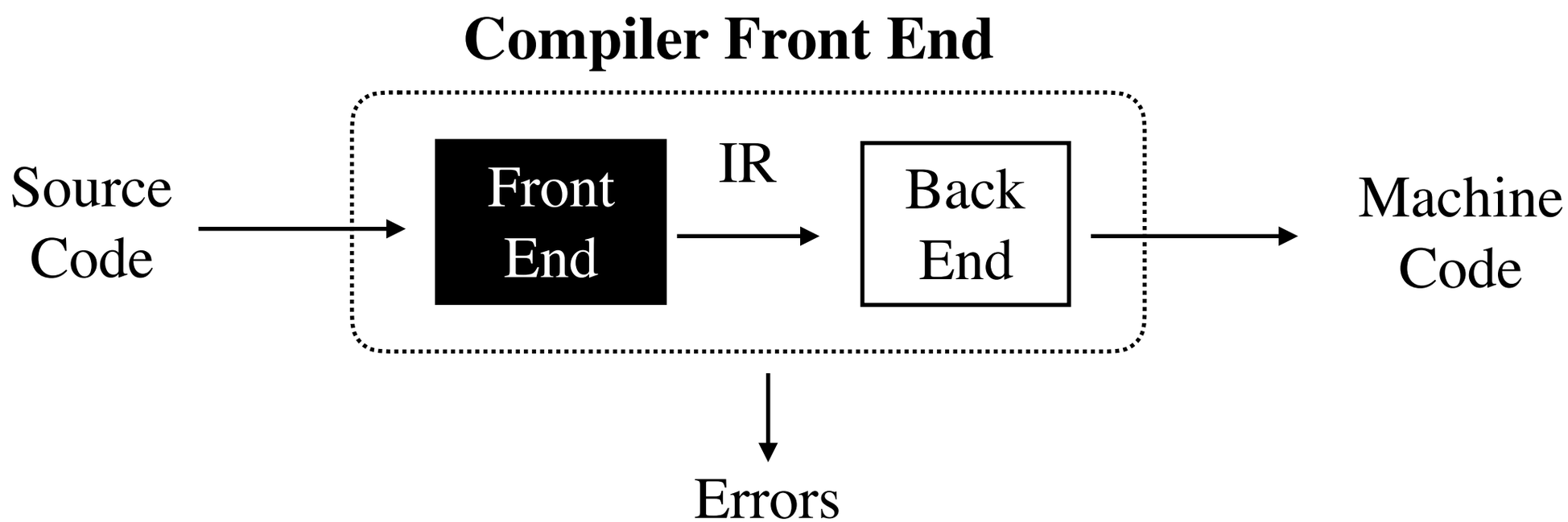
Example IR: Abstract Syntax Tree

Compilers often use an abstract syntax tree.

x + 2 - y



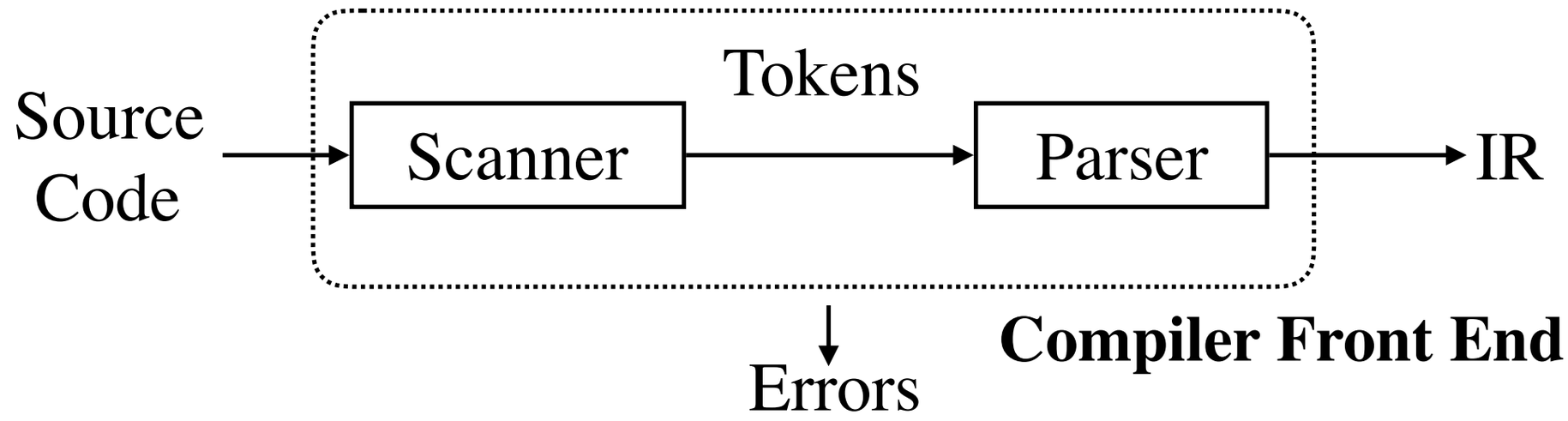
Traditional Two-Pass Compilers



Implications:

- Intermediate representation (IR)
- Front end maps legal code into IR
- Back end maps IR onto target machine
- Simplify retargeting
- Allows multiple front ends and back ends

Front End



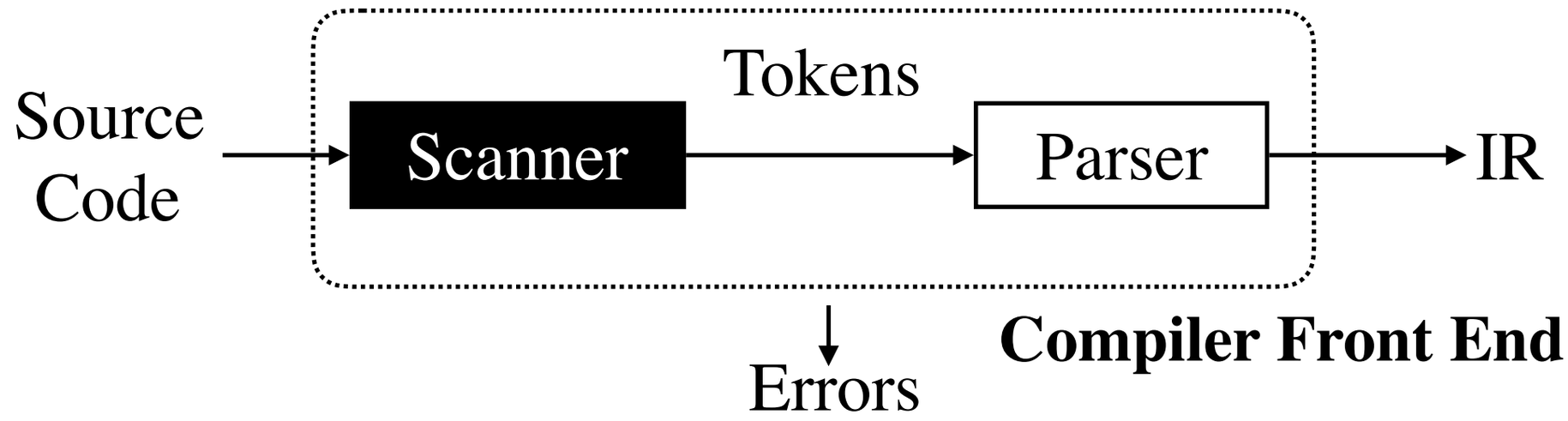
Syntax & semantic analyzer, IR code generator

Front End Responsibilities:

- Recognize legal programs
- Report errors
- Produce IR
- Preliminary storage map
- Shape the code for the back end

Much of front end construction can be automated

Scanner



Scanner

- Maps characters into tokens — the basic unit of syntax

  x = x + y;

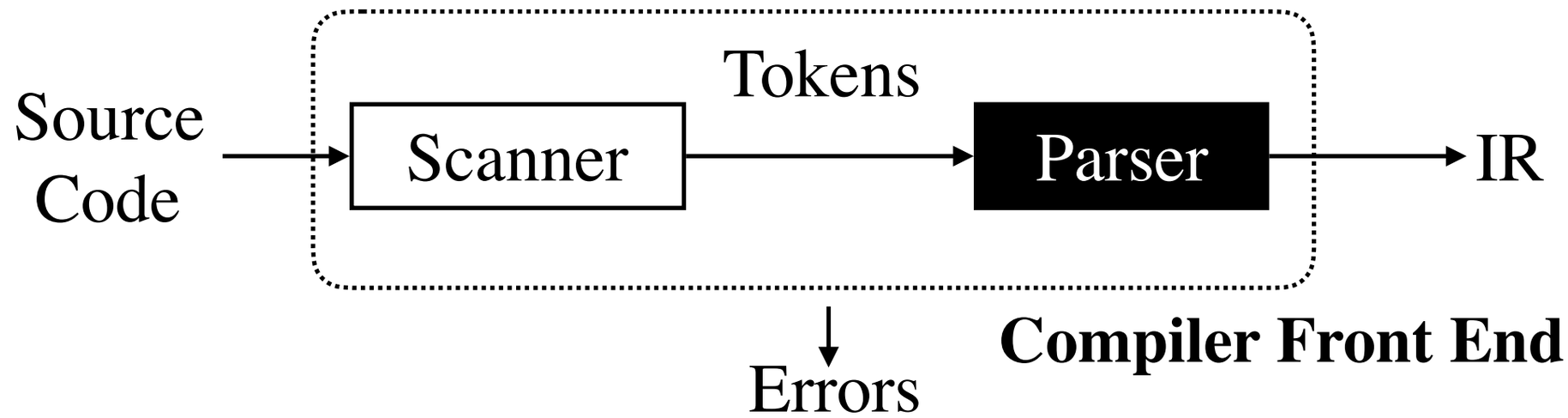
  becomes

  <id, x> = <id, x> + <id, y> ;

- Character string for a token is a lexeme
- Typical tokens: number, id, +, -, *, /, do, end
- Eliminates white space (tabs, blanks, comments)
- A key issue is speed

  ⇒ use specialized recognizer (lex)

Parser



Parser:

- Recognize context-free syntax (Context Free Grammars)
- Guide context-sensitive analysis
- Construct IR(s)
- Produce meaningful error messages
- Attempt error correction

Parser generators mechanize much of the work

Syntax and Semantics of Programming Languages

Syntax:

Describes what a legal program looks like

Semantics:

Describes what a legal program means

Formal Language Definition:

A formal language L is a (possibly *infinite*) set of sentences (*finite* sequences of symbols) over a finite alphabet Σ of (terminal) symbols: $L \subseteq \Sigma^*$

Examples:

- L = { identifiers of length 2} with Σ = {a, b, c}
- L = {strings of only 1s and only 0s}
- L = {strings starting with **$** and ending with #, and any combination of 0s and 1s in between }
- L = {all syntactically correct Java programs}

Syntax and Semantics: How does it work?

Syntactic representation of “values”

What do the following syntactic expressions have in common?

V

101

$ ||||| #

3+ 19 − 17

Syntax and Semantics: How does it work?

Syntactic representation of "values"

What do the following syntactic expressions have in common?

V
101
$ ||||| #
3+ 19 − 17

Answer: They are possible representations of the integer value "5" (written as a decimal number)

What is computation?

Possible answer: A (finite) sequence of syntactic manipulations of value representations ending in a "*normal form*" which is called the result. *Normal forms* cannot be manipulated any further.

Syntax and Semantics: How does it work?

Here is a "game" (**rewrite system**):

Input: Sequence of characters starting with **$** and ending with #, and any combination of **0**s and **1**s in between.

Rules: You may replace a character pattern **X** at any position within the character sequence on the left-hand-side by the pattern **Y** on the right-hand-side: **X⇒Y**:

Rule 1 **$1 ⇒ 1&**

Rule 2 **$0 ⇒ 0$**

Rule 3 **&1 ⇒ 1$**

Rule 4 **&0 ⇒ 0&**

Rule 5 **$# ⇒ →A**

Rule 6 **&# ⇒ →B**

Replace patterns using the rules as often as you can.
When you cannot replace a pattern any more, stop.

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

0 **$ 0** **#** is rewritten as **0** **0 $** **#** by rule 2

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

0 **$ 0** **#** is rewritten as **0** **0 $** **#** by rule 2

0 0 **$ #** is rewritten as **0 0** **→A** by rule 5

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

0 **$ 0** **#** is rewritten as **0** **0 $** **#** by rule 2

0 0 **$ #** is rewritten as **0 0** **→A** by rule 5

no more rules can be applied (**STOP**)

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

0 **$ 0** **#** is rewritten as **0** **0 $** **#** by rule 2

0 0 **$ #** is rewritten as **0 0** **→A** by rule 5

no more rules can be applied (**STOP**)

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

More examples:

$ 0 1 1 0 1 #

$ 1 0 1 0 0 #

$ 1 1 0 0 1 #

Questions:

- Can we get different "results" for the same input string?
- Does all this have a meaning (**semantics**), or are we just pushing symbols?

Syntax and Semantics: How does it work?

Example input:

$ 0 0 #

$ 0 **0 #** is rewritten as **0 $** **0 #** by rule 2

0 **$ 0** **#** is rewritten as **0** **0 $** **#** by rule 2

0 0 **$ #** is rewritten as **0 0** **→A** by rule 5

no more rules can be applied (**STOP**)

Rule 1	**$1 ⇒ 1&**
Rule 2	**$0 ⇒ 0$**
Rule 3	**&1 ⇒ 1$**
Rule 4	**&0 ⇒ 0&**
Rule 5	**$# ⇒ →A**
Rule 6	**&# ⇒ →B**

More examples:

$ 0 1 1 0 1 # →B

$ 1 0 1 0 0 # →A

$ 1 1 0 0 1 # →B

Questions:

- Can we get different "results" for the same input string?
- Does all this have a meaning (**semantics**), or are we just pushing symbols?

Syntax and Semantics of Prog. Languages

The syntax of programming languages is often defined in two layers:
Tokens* and *Sentences

- T*okens - basic units of the language*

Question: How to spell a token (word)?

Answer: Regular expressions

- S*entences - legal combinations of tokens in the language*

Question: How to build correct sentences with tokens?

Answer: (context - free) grammars (CFG)

- E.g., Backus-Naur form (BNF) is a formalism used to express the syntax of programming languages.

Formalisms for Lexical and Syntactic Analysis

Two issues in ***Formal Languages:***

- *Language Specification* → formalism to describe what a valid program (sentence) looks like.

- *Language Recognition* → formalism to describe a machine and an algorithm that can verify that a program is valid or not.

Formalisms for Lexical and Syntactic Analysis

Two issues in ***Formal Languages:***

- *Language Specification* → formalism to describe what a valid program (sentence) looks like.

- *Language Recognition* → formalism to describe a machine and an algorithm that can verify that a program is valid or not.

1. Lexical Analysis: Converts source code into sequence of tokens.
   Regular grammar/expression to specify.
   Finite automata to recognize.

2. Syntax Analysis: Structures tokens into parse tree.
   Context-free grammars to specify.
   Push-down automata to recognize

Lexical Analysis (Scott 2.1, 2.2)

Character sequence

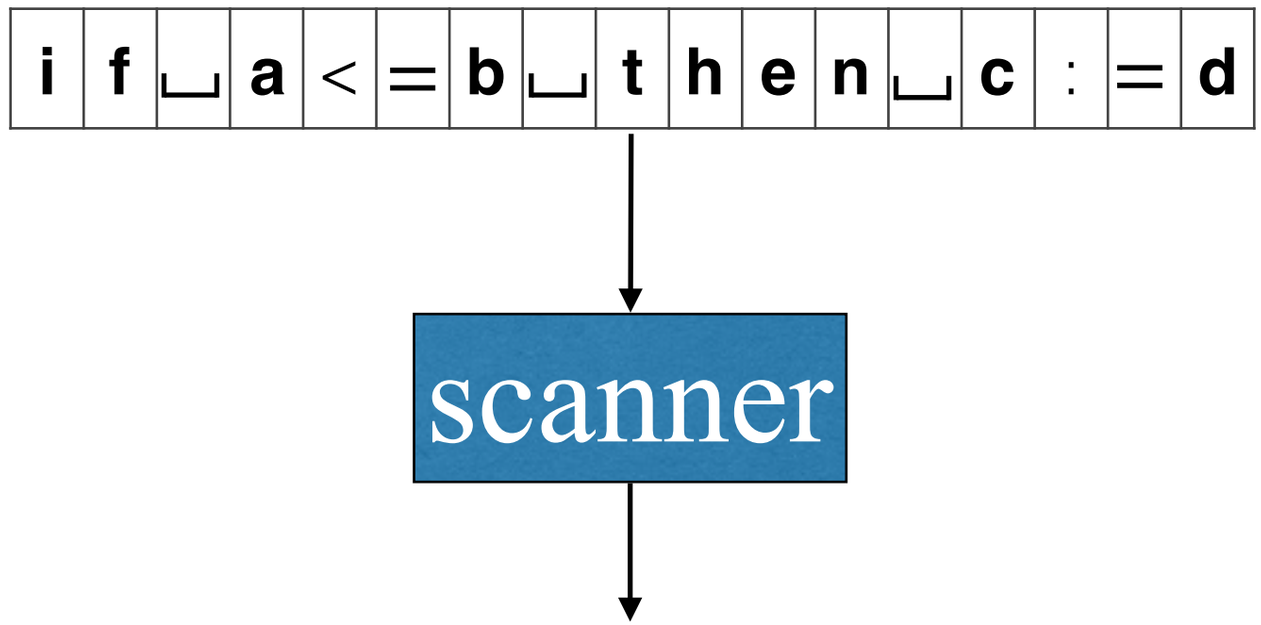
i f a < = b t h e n c : = d

Lexical Analysis (Scott 2.1, 2.2)

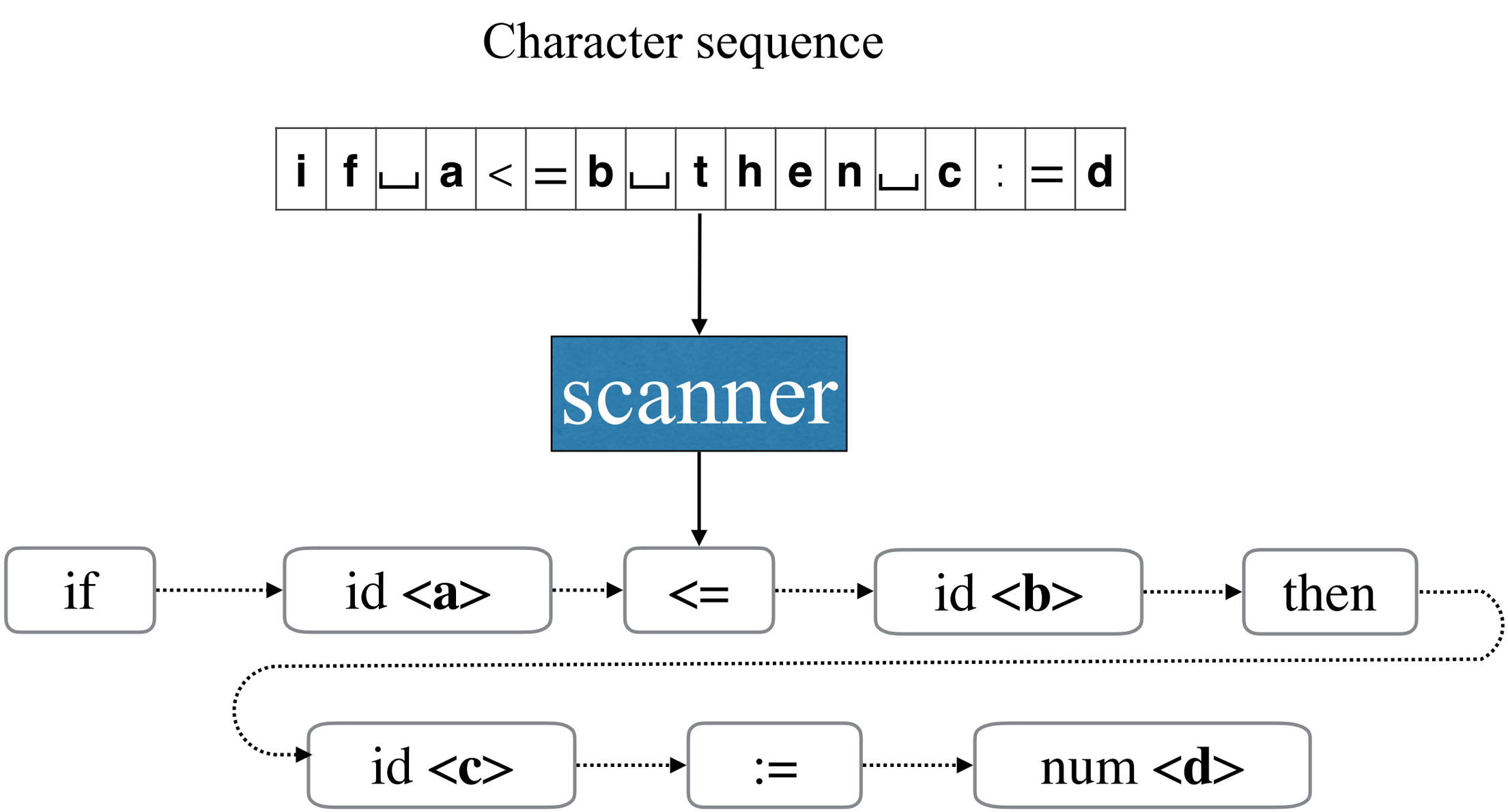
Character sequence

i	f	␣	a	<	=	b	␣	t	h	e	n	␣	c	:	=	d

Lexical Analysis (Scott 2.1, 2.2)



Lexical Analysis (Scott 2.1, 2.2)



Lexical Analysis (Scott 2.1, 2.2)

Tokens (*Terminal Symbols of CFG, Words of Lang*.)

- Smallest "atomic" units of syntax
- Used to build all the other constructs
- Example, **C**:

keywords: **for if goto volatile**…

= * / - < > == <= >= <> ( ) [ ] ; := . , …

number: (Example: 3.14 28 …)

identifier: (Example: b square addEntry …)

Lexical Analysis (cont.)

Identifiers

- Names of variables, etc.
- Sequence of terminals of restricted form;
  Example, in C language: **A31,** but not **1A3**
- Upper/lower case sensitive?

Keywords

- Special identifiers which represent tokens in the language
- May be reserved (reserved words) or not
  - E.g., C: "**if**" is reserved.

Delimiters – When does character string for token end?

- Example: identifiers are longest possible character sequence that does not include a delimiter.
- Most languages have more delimiters such as '␣', new line, keywords, ...

Regular Expressions

A syntax (notation) to specify regular languages.

RE r	**Language L(r)**
a	**{a}**
ϵ	{**ϵ**}

Regular Expressions

A syntax (notation) to specify regular languages.

RE r	**Language L(r)**
r | s	L(r) ∪ L(s)
rs	{RS | R ∈ L(r), S ∈ L(s)}
r_+	L(r) ∪ L(rr) ∪ L(rrr) ∪…
r* (r* = r^+ | ϵ)	{ϵ} ∪ L(r) ∪ L(rr) ∪ L(rrr) ∪... (any number of r's concatenated)
(s)	L(s)

Examples of Expressions— Solution

RE	Language
a|bc	
(b|c)a	
a ϵ	
a*|b	
ab*	
ab*|c^{+}	
(a|b)*	
(0|1)*0	

Examples of Expressions— Solution

RE	Language
a|bc	**{a, bc}**
(b|c)a	
a ϵ	
a*|b	
ab*	
ab*|c^+	
(a|b)*	
(0|1)*0	

Examples of Expressions— Solution

RE	Language
a|bc	**{a, bc}**
(b|c)a	**{ba, ca}**
a ϵ	
a*|b	
ab*	
ab*|c^{+}	
(a|b)*	
(0|1)*0	

Things to Do

Things to do for next lecture:

- Read Scott: Chapter 1
- Read Scott: Chapters 2.1 and 2.2; ALSU: Chapters 3.1 - 3.4
- Get an ilab account
- Learn to use Sakai discussion group

Recitations will start **Next Week**.