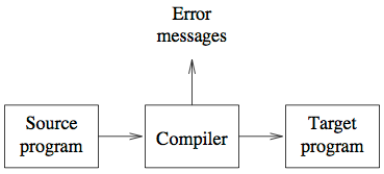
CSCE 3600
Principles of Systems Programming

Compilers and Compilation

University of North Texas

Compiler Execution

Compilation

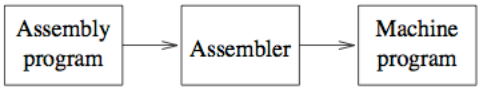


- A process that translates a program in one language (the source language) into an *equivalent* program in another language (the object or target language)
  - Important part is detection and reporting of errors
  - Source language usually high-level programming language (i.e., problem-oriented language)
  - Target language is a machine language or assembly language (i.e., machine-oriented language)
- Compilation is the link between abstract world of application development and low-level world of application execution on machines

3

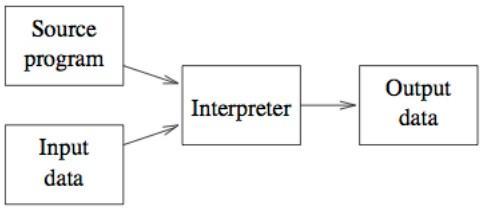
Types of Translators

- Assembler is also a type of translator



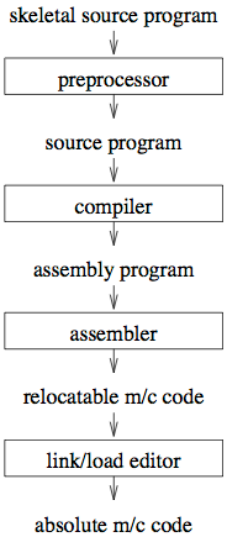
- Interpreter closely related to compiler, but takes both source program and input data
  - Translation and execution phases of the source program are the same

- Easier implementation of programs (run-time errors immediately displayed)
- Slower execution
- Often requires more space



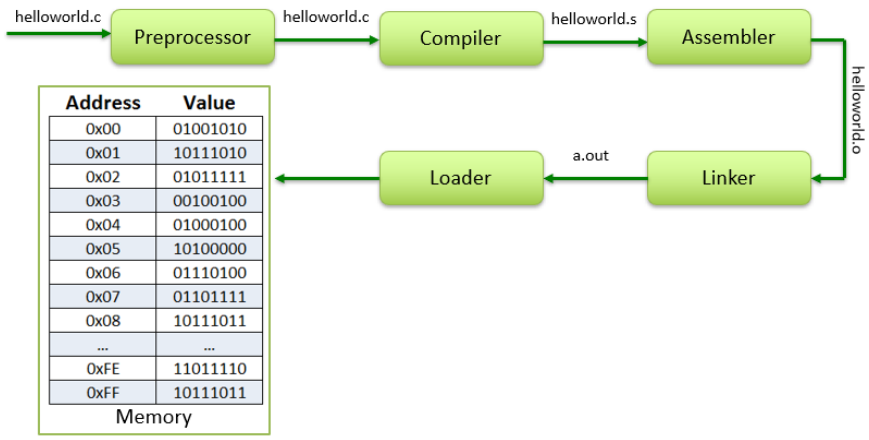
4

Program Compilation Process



5

Program Compilation Process



6

C Preprocessor

- Preprocessing
  - Occurs before a program is compiled
  - Inclusion of other files
  - Definition of symbolic constants and macros
  - Conditional compilation of program code
  - Conditional execution of preprocessor directives
- Format of preprocessor directives
  - Lines begin with **#**
  - Only whitespace characters before directives on a line
- Example

```
gcc -E helloworld.c
```

  - Stop after preprocessing and send output to `stdout`

7

#include Preprocessor Directive

- Copy of a specified file included in place of directive

`#include <filename>`

  - Searches standard library for file
  - Use for standard library files

`#include "filename"`

  - Searches current directory, then standard library
  - Use for user-defined files
- Used for
  - Programs with multiple source files compiled together
  - Header file has common declarations and definitions (classes, structures, function prototypes)
    - **`#include`** statement in each file

8

#define Preprocessor Directive

- Symbolic constants
  - When program compiled, all occurrences of symbolic constant replaced with replacement text
  - Format

    `#define` *identifier replacement-text*
  - Example:

    `#define PI 3.14159`
  - Everything to right of identifier replaces text

    `#define PI = 3.14159`
    - Replaces "**`PI`**" with "**`= 3.14159`**"
  - Cannot redefine symbolic constants once created

9

#define Preprocessor Directive

- Macro
  - Macro without arguments treated like a symbolic constant
  - A macro with arguments has its arguments substituted for replacement text, when the macro is expanded
  - Performs a text substitution – no data type checking
  - The macro

```
#define CIRCLE_AREA( r ) ( PI * ( r ) * ( r ) )
```

would cause

```
area = CIRCLE_AREA( 4 );
```

to become

```
area = ( 3.14159 * ( 4 ) * ( 4 ) );
```

10

#define Preprocessor Directive

- Use parentheses
  - Without them the macro

```
#define CIRCLE_AREA( x )  PI * ( x ) * ( x )
```

would cause

```
area = CIRCLE_AREA( c + 2 );
```

to become

```
area = 3.14159 * c + 2 * c + 2;
```

- Multiple arguments

```
#define RECTANGLE_AREA( x, y )  ( ( x ) * ( y ) )
```

would cause

```
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

to become

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

11

#undef Preprocessor Directive

- Undefines a symbolic constant or macro
- If a symbolic constant or macro has been undefined, it can later be redefined
- Example

```
#define WIDTH 80
#define ADD( X, Y ) ((X) + (Y))
.
.
.
#undef WIDTH
#undef ADD
```

12

Conditional Compilation

- Control preprocessor directives and compilation
- Cast expressions, **sizeof**, enumeration constants cannot be evaluated in preprocessor directives
- Structure similar to **if**

```
#if !defined( NULL )
    #define NULL 0
#endif
```

  - Determines if symbolic constant **NULL** has been defined
    - If **NULL** is defined, **defined(NULL)** evaluates to **1**
    - If **NULL** is not defined, this function defines **NULL** to be **0**
- Every **#if** must end with **#endif**
- **#ifdef** short for **#if defined(name)**
- **#ifndef** short for **#if !defined(name)**

13

Conditional Compilation

- Other statements
  - **#elif** – equivalent of **else if** in an **if** structure
  - **#else** – equivalent of **else** in an **if** structure
- "Comment out" code
  - Cannot use **/*** ... ***/**
  - Use

```
#if 0
    code commented out
#endif
```

  - To enable code, change **0** to **1**

14

Compiler

- GNU Compiler Collection (gcc) is standard Linux C compiler (see `man gcc` for details)
- Compiler translates C program to assembler code
- Example

```
gcc –S helloworld.c
```

  - Stop after stage of compiling proper (i.e., not passed to assembler) and outputs assembler code to **`helloworld.s`**
- Compiler actually consists of many parts (more later)
  - Parser
  - Analyzer
  - Code optimizer
  - Code generation

15

Assembler

- Assembles, but does not link, source code to generate relocateable object code
  - Object code may contain metadata such as label definitions that refer to locations within sections of code
  - May include holes (i.e., relocation entries) that are to be filled with the values of labels defined elsewhere
- Example

```
gcc –c helloworld.c
```

  - Assemble source code and save object file as **`helloworld.o`**
- Use **`nm`** or **`objdump`** to look at object

```
nm example.o
objdump –t example.o
```

16

Linker

- Combines previous compiled and assembled object code along with standard library functions to make executable file
  - Resolves references in each object file to external variables and procedures declared in other files
- Example

```
gcc helloworld.o
```

  - Build default executable `a.out`
  - Use `gcc -lm helloworld.o` to link math library
- Use `nm` to look at executable

17

Loader

- Compilers, assemblers, and linkers usually produce code whose memory references are made *relative* to an undetermined starting location that can be anywhere in memory (relocatable machine code)
- The *loader* ensures that the object programs are placed in memory in an executable form
  - Loader calculates appropriate absolute addresses for these memory locations and amends the code the use these addresses
    - Gets an address from the OS to place the program in memory
    - Changes necessary addresses (if any)
    - Places code into memory to run

18

Compiling and Linking

- Only compiling (creating hello.o)
  gcc –c hello.c
- Only linking
  gcc hello.o –o hello
- Compiling and linking
  gcc hello.c –o hello
- To run your program
  ./hello
  Hello World!
- Additional common flags to gcc

–g	allows debugging
–l<library_name>	linking with external libraries

If you leave off the –o, executable goes into the a.out file

19

Compiler Construction

20

What is a Compiler?

- All computers only understand machine language

- Therefore, high-level language instructions must be translated into machine language prior to execution

21

What is a Compiler?

- Consists of a piece of system software that translates high-level languages into machine language

```
while (c !='x')
{
    if (c == 'a' || c == 'e' || c == 'i')
        printf("Congrats!");
    else
        if (c!='x')
            printf("Sorry!");
}
```

program.c

Compiler

gcc -o prog program.c

22

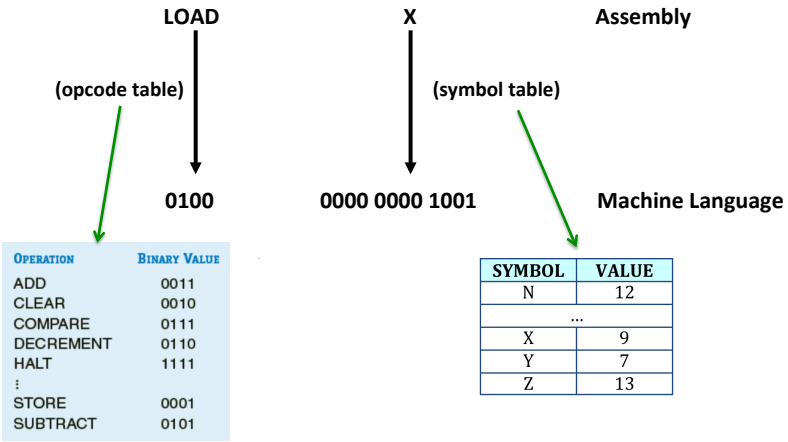
Why Build Compilers?

- Compilers provide an essential interface between applications and architectures
- High level programming languages:
  - Increase programmer productivity
  - Better maintenance
  - Portable
- Low level machine details:
  - Instruction selection
  - Addressing modes
  - Pipelines
  - Registers and cache
- Compilers efficiently bridge the gap and shield the application developers from low level machine details

23

Assembler

- A kind of compiler
  - One-to-one translation



24

Compiler

- A high-level language translator
  - One-to-many translation

a = b + c - d;


```
0100 00001110001    LOAD B
0011 00001110010    ADD C
0101 00001110011    SUBTRACT D
0001 00001110100    STORE A
```

25

Compiler Properties

- Compiler must generate a correct executable
  - The input program and the output program must be equivalent, the compiler should preserve the meaning of the input program
- Output program should run fast
  - For optimizing compilers, we expect the output program to be more efficient than the input program
- Compiler itself should be fast
- Compiler should provide good diagnostics for programming errors
- Compiler should support separate compilation
- Compiler should work well with debuggers
- Optimizations should be consistent and predictable
- Compile time should be proportional to code size

26

Compiler Goals

- Code produced must be correct

A = (B + C) – (D + E);

Possible translation:

```
LOAD B
ADD C
STORE B
LOAD D
ADD E
STORE D
LOAD B
SUBTRACT D
STORE A
```

Is this correct?

No - STORE B and STORE D change the values of variables B and D which is the high-level language does not intend

27

Compiler Goals

- Code produced should be reasonably efficient and concise

Compute the sum: $2x_1 + 2x_2 + 2x_3 + 2x_4 + \ldots + 2x_{50000}$

```
sum = 0.0
for (i = 0; i < 50000; i++)
    sum = sum + (2.0 * x[i]);
```

Optimizing compiler:

```
sum = 0.0
for (i = 0; i < 50000; i++)
    sum = sum + x[i];
sum = sum * 2.0;
```

49,999 less instructions

28

Why Study Compilers?

- Compilers embody a wide range of theoretical techniques and their application to practice
  - DFAs, DPDAs, formal languages, formal grammars, lattice theory
- Compiler construction teaches programming and software engineering skills
- Compiler construction involves a variety of areas
  - Theory, algorithms, systems, architecture
- The techniques used in various parts of compiler construction are useful in a wide variety of applications
  - Many practical applications have embedded languages, commands, macros, etc.
- Is compiler construction a solved problem?
  - No! New developments in programming languages and machine architectures (multicore machines) present new challenges

29

Syntax & Semantics of Language

- A programming language must include the specification of syntax (structure) and semantics (meaning)
- Syntax typically means the context-free syntax because of the almost universal use of context-free grammars (CFGs)
- Example
  - `a = b + c` is syntactically legal
  - `b + c = a` is illegal

CFG is a set of recursive rewriting rules used to generate patterns of strings

- The *semantics* of a programming language are commonly divided into two classes
  - Static semantics
    - Semantics rules that can be checked at compile time
      - E.g., the type and number of a function's argument(s)
  - Runtime semantics
    - Semantics rules that can be checked only at run time

30

Compilation

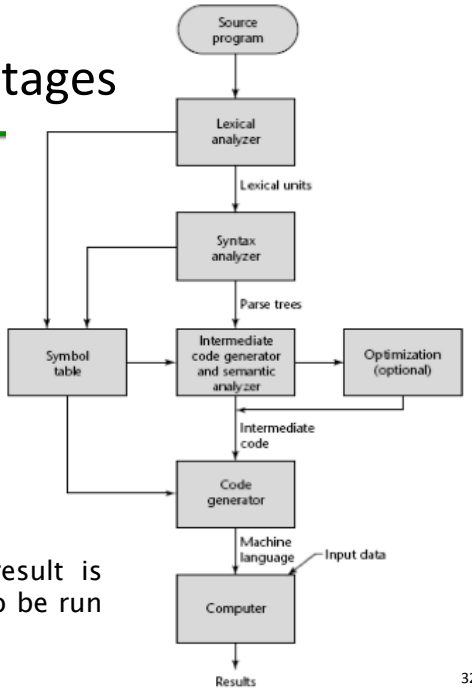
- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
  - Lexical Analysis
    - Converts characters in source program into lexical units
  - Syntax Analysis
    - Transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Semantics Analysis
    - Generate intermediate code
  - Code Generation
    - Machine code is generated

31

Compilation Stages

machine independent

- Scanner
- Parser
- Semantic analysis
- Intermediate code generation
- Machine-independent code improvement (optional)
- Target code generation
- Machine-specific code improvement (optional)

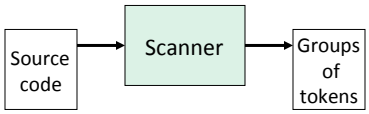


- For many compilers, the result is assembly, which then has to be run through an assembler

32

Lexical Analysis

- Lexical analyzer or scanner
  - Recognizes the "tokens" of a program
  - The compiler scans the source code from left-to-right, character-by-character, and groups these characters into lexemes (sequence of characters that match a pattern) and outputs a sequence of tokens to the syntax analyzer
  - Each token represents a logically cohesive sequence of characters
    - Keywords
      `if, while, ...`
    - Operators
      `+, *, <=, ||, ...`
    - Identifiers (names of variables, arrays, procedures, classes)
      `i, i1, j1, count, sum, ...`
    - Numbers
      `12, 3.14, 7.2E-2, …`



33

Lexical Analysis

- The main functions of this phase are
  - Identify the lexical units in a source statement
  - Classify units into different lexical classes (e.g., constants, reserved words, etc.) and enter them in different tables
  - Build a descriptor (called a token) for each lexical unit (group of input characters)
  - Ignore comments in the source program
  - Detect tokens that are not a part of the language

Tokens are syntactical units that are treated as single, indivisible entities for the purposes of translation

34

Lexical Analysis Example

- Program statement

  `sum = sum + a[i]`

- Digital perspective:

  `tab,s,u,m,blank,=,blank,s,u,m,blank,+,blank,a,[,i,],;`

- Tokenized:

  `sum,=,sum,+,a[i],;`

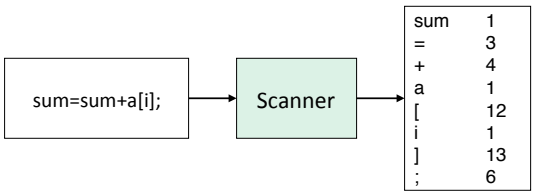
TOKEN TYPE	CLASSIFICATION NUMBER
Symbol	1
Number	2
=	3
+	4
-	5
;	6
==	7
if	8
else	9
(	10
)	11
[	12
]	13
…	

Typical Token Classifications

35

Lexical Analysis Process

1. Discard blanks, tabs, etc. – look for beginning of token
2. Put characters together
3. Repeat step 2 until end of token
4. Classify and save token
5. Repeat steps 1 – 4 until end of statement
6. Repeat steps 1 – 5 until end of source code



36

How Do We Specify Lexical Patterns?

- Some patterns are easy
  - Keywords and operators
    - Specified as literal patterns: if, then, else, while, =, +, …
- Some patterns are more complex
  - Identifiers
    - Letter followed by letters and digits
  - Numbers
    - Integer: 0 or a digit between 1 and 9 followed by digits between 0 and 9
    - Decimal: An optional sign which can be "+" or "-" followed by digit "0" or a nonzero digit followed by an arbitrary number of digits followed by a decimal point followed by an arbitrary number of digits

We want to have concise descriptions of patterns, and we want to automatically construct the scanner from these descriptions

37

Language Definitions

- An *alphabet* Σ is a finite non-empty set (of symbols)
- A *string* or *word* over an alphabet Σ is a finite *concatenation* of symbols from Σ
- The *length* of a string w (i.e., number of characters comprising the string) is denoted |w|
- The *empty* or *null* string is denoted ε (i.e., |ε|=0)
- The set of all strings over Σ is denoted Σ*
- For each $n \geq 0$, we define $\Sigma^n = \{w \in \Sigma^* \mid |w| = n\}$
- For a symbol or word x, x^n denotes x concatenated with itself n times, with the convention that x^0 denotes ε
- A language over Σ is a set $L \subseteq \Sigma^*$
- Two languages L_1 and L_2 over common alphabet Σ are equal if they are equal as sets

38

Lexical Patterns: Regular Expressions

- Regular expressions (regex) describe regular languages
- Regular Expression (over alphabet Σ)
  - ε (empty string) is a regex denoting the set {ε}
  - If **a** is in Σ, then **a** is a regex denoting {**a**}
  - If x and y are regex denoting languages L(x) and L(y) then
    - x is a regex denoting L(x)
    - x | y is a regex denoting L(x) ∪ L(y)
    - xy is a regex denoting L(x)L(y)
    - x^* is a regex denoting L(x)*

Precedence is *closure*, then *concatenation*, then *alternation*

All **left-associative**

$x \mid y^* z$ is equivalent to
$x \mid ((y^*) z)$

39

Regular Expression Examples

- All strings of 1s and 0s
  $(0 \mid 1)^*$
- All strings of 1s and 0s beginning with a 1
  $1 (0 \mid 1)^*$
- All strings of 0s and 1s containing at least two consecutive 1s
  $(0 \mid 1)^* 1 1 (0 \mid 1)^*$
- All strings of alternating 0s and 1s
  $(\varepsilon \mid 1) (0 1)^* (\varepsilon \mid 0)$

40

Regular Expression Extensions (Lex)

- $x+ = x\,x^*$ — denotes $L(x)^+$
- $x? = x \mid \varepsilon$ — denotes $L(x) \cup \{\varepsilon\}$
- [abc] = a | b | c — matches one character in the square bracket
- a-z = a | b | c | ... | z — range
- [0-9a-z] = 0 | 1 | 2 | ... | 9 | a | b | c | ... | z
- [^abc] — ^ means negation; matches any character except a, b or c
- . — (dot) matches any character except the newline
- . = [^\n] — \n means newline, dot is equivalent to [^\n]
- "[" — matches left square bracket, metacharacters in double quotes become plain characters
- \[— matches left square bracket, metacharacter after backslash becomes plain character

41

Regular Definitions

- We can define macros using regular expressions and use them in other regular expressions

  Letter → (a|b|c| … |z|A|B|C| … |Z)

  Digit → (0|1|2| … |9)

  Identifier → *Letter* (*Letter* | *Digit*)*
- Important: We should be able to order these definitions so that every definition uses only the definitions defined before it (i.e., no recursion)
- Regular definitions can be converted to basic regular expressions with macro expansion
- In Lex, enclose definitions using curly braces

  Identifier → {*Letter*} ({*Letter*} | {*Digit*})*

42

Regular Expression Examples

Digit → (0|1|2| … |9)

Integer → (+|-)? (0| (1|2|3| … |9)(*Digit* *))

Decimal → *Integer "." Digit* *

Real → (*Integer* | *Decimal*) E (+|-)?*Digit* *

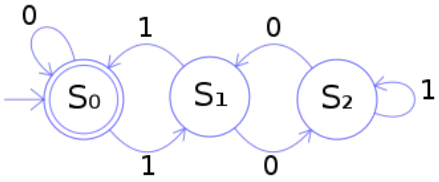
Complex → *"* (*" Real , Real "*) *"*

Numbers can get even more complicated.

43

Regular Expression to Scanners

- Regular expressions are useful for specifying patterns that correspond to tokens
- However, we also want to construct programs that recognize these patterns
- How do we do it?
  - Use finite automata!



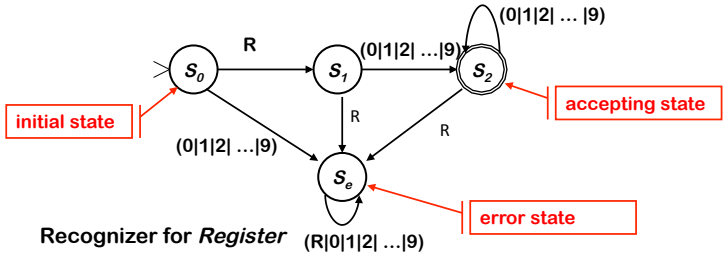
44

RegEx to DFA Example

- Consider the problem of recognizing register names in an assembler

  Register → R (0|1|2| … |9) (0|1|2| … |9)*
- Allows registers of arbitrary number
- Requires at least one digit
- REGEX corresponds to a recognizer (or DFA)

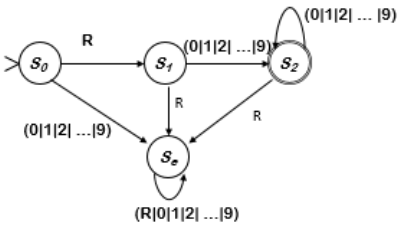
A recognizer tells whether a given string "belongs to" a grammar



45

Deterministic Finite Automata (DFA)

- A set of states *S*
  - $S = \{ s_0, s_1, s_2, s_e \}$
- A set of input symbols (an alphabet) Σ
  - $\Sigma = \{ R, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$
- A transition function $\delta : S \times \Sigma \rightarrow S$
  - Maps (state, symbol) pairs to states
  - $\delta = \{ (s_0, R) \rightarrow s_1, (s_0, 0\text{-}9) \rightarrow s_e, (s_1, 0\text{-}9) \rightarrow s_2, (s_1, R) \rightarrow s_e,$
    $(s_2, 0\text{-}9) \rightarrow s_2, (s_2, R) \rightarrow s_e, (s_e, R \mid 0\text{-}9) \rightarrow s_e \}$
- A start state
  - s_0
- A set of final (or accepting) states
  - $Final = \{ s_2 \}$

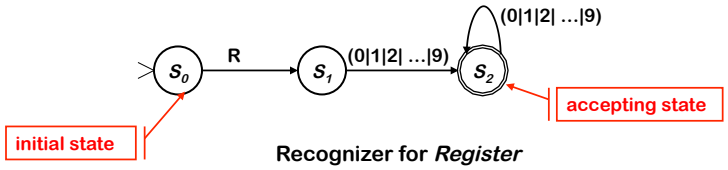


A DFA accepts a word x iff there exists a path in the transition graph from start state to a final state such that the edge labels along the path spell out x

46

DFA Simulation

- Start in state s_0 and follow transitions on each input character
- DFA accepts a word *x iff* *x* leaves it in a final state (s_2)



Recognizer for *Register*

So,

- "R17" takes it through s_0, s_1, s_2 and accepts
- "R" takes it through s_0, s_1 and fails
- "A" takes it straight to s_e
- "R17R" takes it through s_0, s_1, s_2, s_e and rejects

47

DFA Simulation

```
state = s0;
char = get_next_char();
while (char != EOF) {
  state = δ(state,char);
  char =get_next_char();
}
if (state ∈ Final)
  report acceptance;
else
   report failure;
```

We can store the transition table in a two-dimensional array:

δ	R	0,1,2,3, 4,5,6, 7,8,9	other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Final = { s_2 }
We can also store the final states in an array

- *The recognizer translates directly into code*
- *To change DFAs, just change the arrays*
- *Takes O(|x|) time for input string x*

48

Recognize Longest Accepted Prefix

```
accepted = false;
current_string = ε;        // empty string
state  = s0;       // initial state
if  (state ∈ Final) {
 accepted_string = current_string;
 accepted = true;
}
char =get_next_char();
while (char != EOF) {
  state = δ(state,char);
  current_string = current_string + char;
  if  (state ∈ Final) {
    accepted_string = current_string;
    accepted = true;
  }
  char =get_next_char();
}
if  accepted
 return accepted_string;
else
   report error;
```

Given an input string, this simulation algorithm returns the **longest accepted prefix**

δ	R	0,1,2,3, 4,5,6, 7,8,9	other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Final = { s_2 }

Given the input "R17R", this simulation algorithm returns "R17"

49

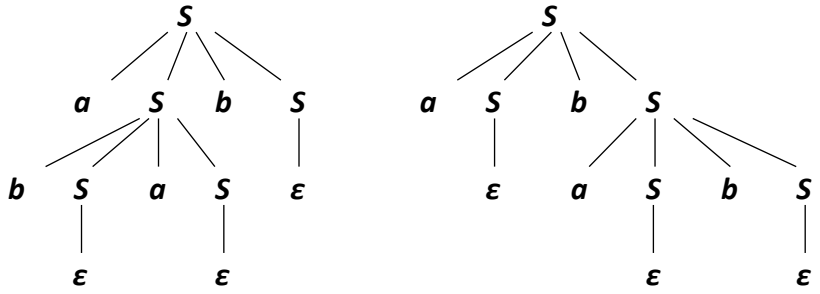
Syntax Analysis

- Syntax analyzer, or parser, groups sequences of tokens from the lexical analysis phase into phrases, each with an associated *phrase type* (i.e., a logical unit with respect to the rules of the source language)
- The following operations are performed in this phase
  - Obtain tokens from lexical analyzer
  - Check whether the expression is syntactically correct
  - Report syntax errors, if any
  - Determine the statement class
    - An assignment statement, a condition (e.g., `if`) statement, etc.
  - Group tokens into statements
  - Construct hierarchical structures called parse trees
    - Parse trees represent the syntactic structure of the program

50

Grammars and Parse Trees

- The grammar $S \rightarrow aSbS \mid bSaS \mid \varepsilon$ generates all strings of *a*'s and *b*'s with the same number of *a*'s as *b*'s
- This grammar is ambiguous: abab has two parse trees



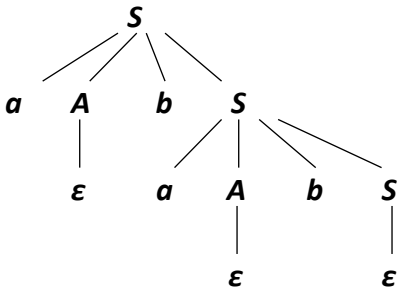
$(ab)^n$ has $\frac{1}{n+1}\binom{2n}{n}$ parse trees

51

Grammars and Parse Trees

- Natural languages are inherently ambiguous
  - But programming languages should not be!
- This grammar G generates the same language

$S \rightarrow aAbS \mid bBaS \mid \varepsilon$
$A \rightarrow aAbA \mid \varepsilon$
$B \rightarrow bBaB \mid \varepsilon$

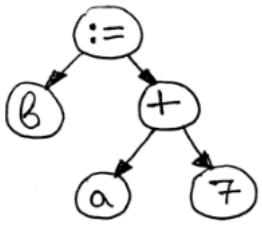


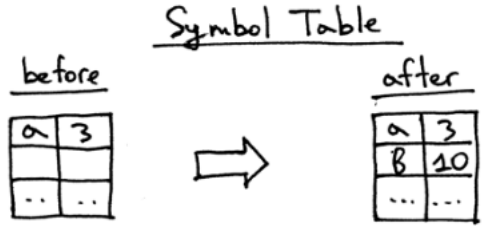
- *G* is unambiguous and has only one parse tree for every sentence in *L*(*G*)

52

Parse Trees

- The statement `b = a + 7` can be represented by the following parse tree
  - Parse tree illustrates the grouping of tokens into phrases
- Successful construction of a parse tree is proof that the statement is correctly formed

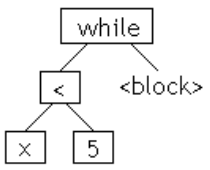




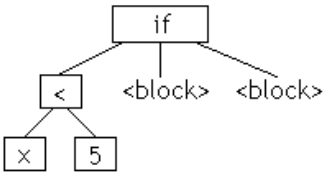
53

Parse Trees

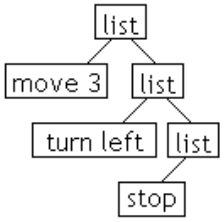
- Every construct (statement, expression, etc.) in a programming language can be represented as a parse tree



Example while loop



Example if-else statement



Example <block>

- Terminals
  - The actual tokens of the language
- Non-Terminals
  - Intermediate grammatical categories used to help explain and organize the language

54

Recursive Descent Parser

- Simplest idea for creating a parser for a grammar is to construct a recursive descent parser
  - Built from a context-free grammar
- Simple rules used to build the recursive program
  - Implement a function for each non-terminal
  - Generate code for each rule
    - "Call" non-terminals
    - "Match" terminals
    - Use `if` statement to choose between multiple rules with the same (non-terminal) symbol

55

Recursive Descent Parser

- Advantages
  - They are exceptionally simple
  - They can be constructed from recognizers simply by doing some extra work – specifically, building a parse tree
- Disadvantages
  - They are not as fast as some other methods
  - It is difficult to provide really good error messages
  - They cannot do parses that require arbitrarily long look-aheads

Recursive descent parsers are great for "quick and dirty" parsing jobs

56

Recursive Descent Parser Stack

- One easy way to do recursive descent parsing is to have each parse method take the tokens it needs, build a parse tree, and put the parse tree on a global stack
  - Write a parse method for each non-terminal in the grammar
    - Each parse method should get the tokens it needs, and *only* those tokens
      - Those tokens (usually) go on the stack
    - Each parse method may call other parse methods, and expect those methods to leave their results on the stack
    - Each (successful) parse method should leave *one* result on the stack

57

Example: while Statement

- <while statement> ::= "while" <condition> <block>
- The parse method for a while statement:
  - Calls the Tokenizer, which returns a "while" token
  - Makes "while" into a tree node, which it puts on the stack
  - Calls the parser for <condition>, which parses a condition and puts it on the stack
    - Stack contains: "while" <condition> ("top" is on right)
  - Calls the parser for <block>, which parses a block and puts it on the stack
    - Stack now contains: "while" <condition> <block>
  - Pops the top three things from the stack, assembles them into a tree, and pushes this tree onto the stack

58

Recognizing a while Statement

```
// <while statement> ::= "while" <condition> <block>
int whileStatement()
{
    if (keyword("while")) {
        if (condition()) {
            if (block()) {
                return 1;  // true
            } else error("Missing '{' ");
        } else  error("Missing <condition>");
    }
    return 0;  // false
}
```

59

Parsing a while Statement

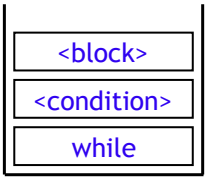
```
// <while statement> ::= "while" <condition> <block>
int whileStatement() {
    if (keyword("while")) {
        if (condition()) {
            if (block()) {
                makeTree(3, 2, 1);
                return 1;  // true
            } else error("Missing '{' ");
        } else  error("Missing <condition>");
    }
    return 0;  // false
}
```

- This code assumes that keyword(String), condition(), and block() all leave their results on a stack
- On the stack, while = 3, <condition> = 2, <block> = 1

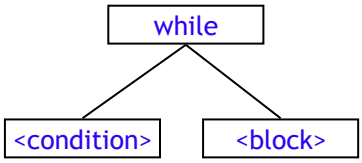
60

Parsing a while Statement

- After recognizing a while loop, the stack looks like this:

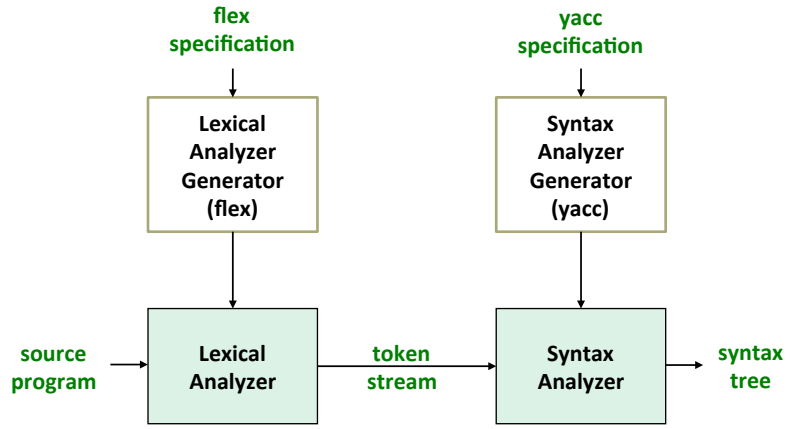



```
void makeTree(int keyword, int left, int right) {
   Tree root = getStackItem(keyword);
   Tree leftChild = getStackItem(left);
   Tree rightChild = getStackItem(right);
   stack.pop();
   stack.pop();
   stack.pop();
   root.addChild(leftChild);
   root.addChild(rightChild);
   stack.push(root);
}
```



61

Compiler Component Generators



62

Desk Calculator Example

- `flex` specification for a desk calculator: **`desk.l`**

  `$ flex desk.l`

- `yacc` specification for a desk calculator: **`desk.y`**

  `$ yacc desk.y`

  – This generates a file called `y.tab.c`, but before compiling this resulting file, you must add the following function prototypes near the top of the file:

```
int yylex(void);
void yyerror(const char *);
```

- Now compile this file with the `flex` and `yacc` libraries

  `$ gcc y.tab.c -ly -lfl`

in public directory

63

Desk Calculator Example (cont'd)

```
$ ./a.out
Enter the expression: 1.8+2.8
Answer: 4.6
Enter:
1.8*10
Answer: 18
Enter:
20.34/.2
Answer: 101.7
Enter:
101.7*.2
Answer: 20.34
Enter:
1.2*(3.4+5.6)
Answer: 10.8
Enter:
^C
```

64

Intermediate Code Generation

- The intermediate code produces a program in a different language, at an intermediate level between the source code and the machine code
  - This allows part of the compiler to be machine independent!
  - Intermediate languages are sometimes assembly languages
- Generation of intermediate code offers following advantages
  - Flexibility
    - A single lexical analyzer/parser can be used to generate code for several different machines by providing separate back-ends that translate a common intermediate language to a machine-specific assembly language
  - Intermediate code is used in interpretation
    - Executed directly rather than translating it into binary code and storing it

65

Semantic Analysis

- A semantic analyzer takes its input from the syntax analysis phase in the form of a parse tree and a symbol table
- Its purpose is to determine if the input has a well-defined meaning
  - In practice, semantic analyzers are mainly concerned with type checking and type coercion based on type rules

66

Semantic Analysis

- The semantic phase has the following functions:
  - Check phrases for semantic errors (e.g., type checking)
    - In a C program, `int x = 10.5;` should be detected as a semantic error
  - Keeps track of types of identifiers and expressions to verify their consistent usage
  - Maintains the symbol table
    - Symbol table contains information about each identifier in a program, such as identifier type, scope of identifier, etc.

67

Semantic Analysis

- The semantic phase has the following functions:
  - Enforces a large number of rules using the symbol table
    - Every identifier is declared before it is used
    - No identifier is used in an inappropriate context (e.g., adding a string to an integer)
    - Function calls have a correct number and type of arguments
  - Checks certain semantic rules, called *dynamic semantics*, at run time
    - Array subscript expression lies within the bounds of the array
    - Variables are never used in an expression unless they have been given a value

68

Symbol Table

- Built and maintained by the semantic analyzer
- Maps each identifier to information known about it
  - Identifier's type (e.g., `int`, `char`, `float`, etc.)
  - Internal structure (if any)
  - Scope (the portion of the program in which it is valid)
- Using the symbol table, the semantic analyzer enforces a large variety of rules
- Purpose of symbol table is to provide quick and uniform access to identifier attributes throughout compilation process

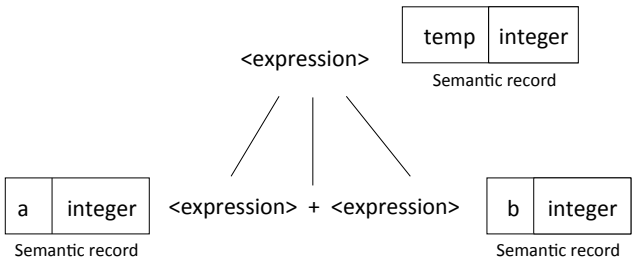
69

Semantics Analysis Example

- Syntactically correct, but semantically incorrect
- Example

```
sum = a + b;
```

Semantic records	
a	integer
sum	double
b	char



70

Code Generation

- The code generated depends upon the architecture of the target machine
  - Knowledge of the instructions and addressing modes in a target computer is necessary for the code generation process
- One of the important aspects of code generation is the efficient initialization of machine resources using assumptions such as
  - Instruction types in target machines
  - Commutative properties of operators in an expression
  - Proper usage of syntax for syntax directed translation

71

Error Handling

- In each phase, a compiler can encounter errors
- On detecting an error, the compiler must
  - Report the error in a helpful way
  - Correct the error, if possible
  - Continue processing (if possible) after the error to look for further errors

72

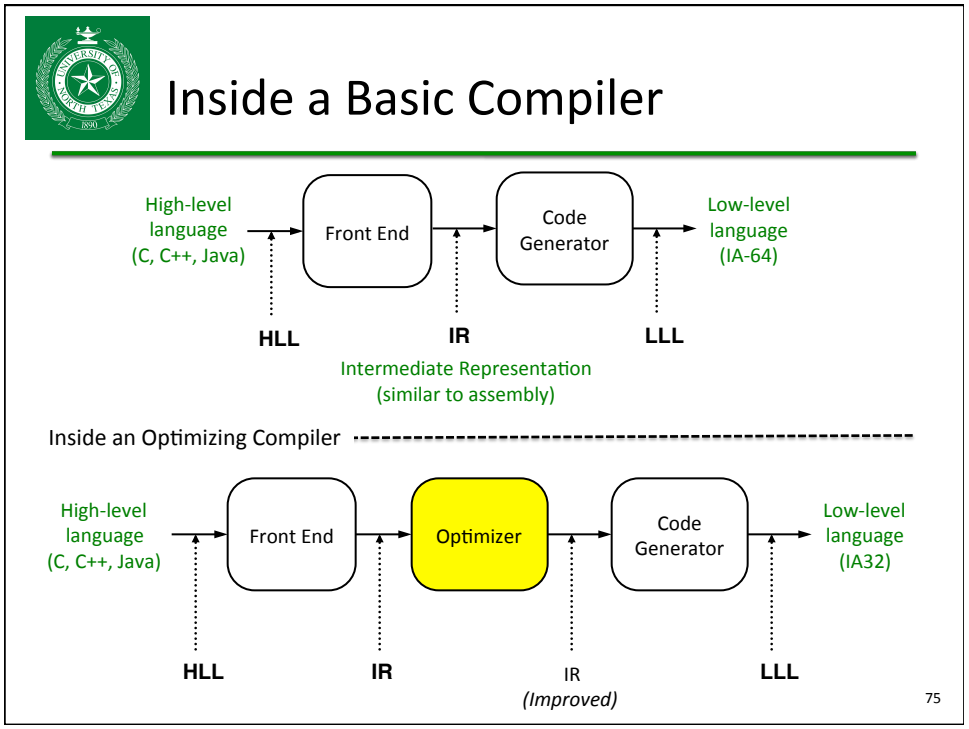
Error Types

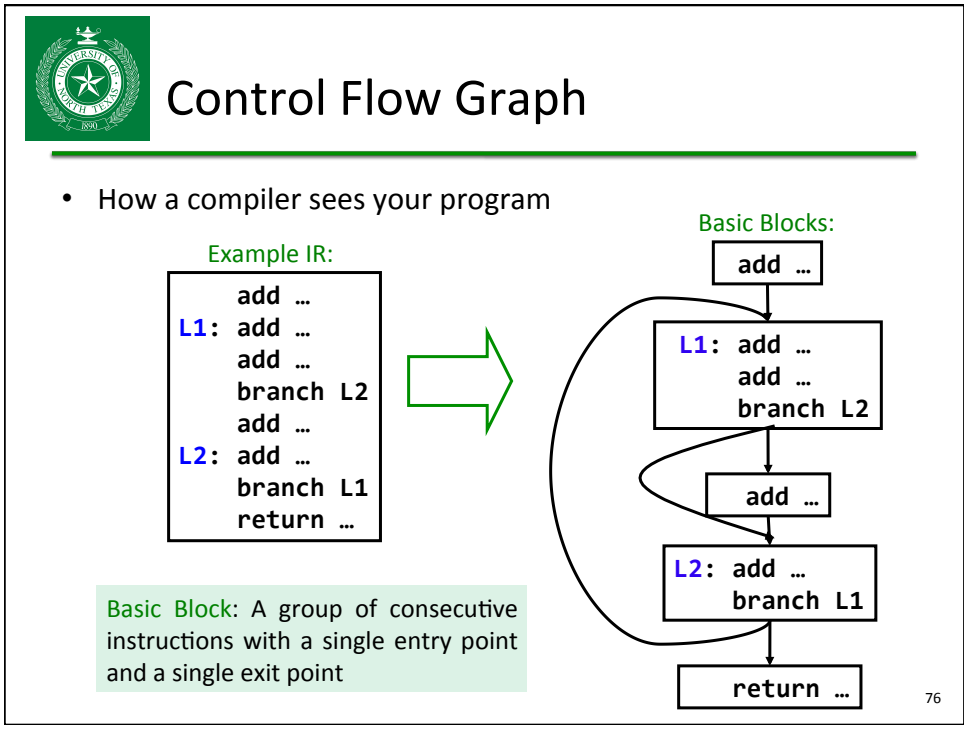
- Syntax errors are errors in the program text
  - A lexical error is a mistake in lexeme (e.g., typing "esle" instead of "else" or missing off one of the quotes in a literal string)
  - A grammatical error is one that violates the rules of the language
- Semantic errors are mistakes concerning the meaning of a program construct
  - Type errors occur when an operator is applied to an argument of the wrong type or to the wrong number of arguments
  - Logical errors occur when a badly conceived program is executed
  - Run-time errors are errors that can be detected only when the program is executed

73

Compiler Optimization

74


Inside a Basic Compiler
High-level language (C, C++, Java)
Front End
Code Generator
Low-level language (IA-64)
HLL
IR
LLL
Intermediate Representation (similar to assembly)
Inside an Optimizing Compiler
High-level language (C, C++, Java)
Front End
Optimizer
Code Generator
Low-level language (IA32)
HLL
IR
IR (Improved)
LLL
75


Control Flow Graph
How a compiler sees your program
Example IR:
add …
L1: add …
add …
branch L2
add …
L2: add …
branch L1
return …
Basic Blocks:
add …
L1: add …
add …
branch L2
add …
L2: add …
branch L1
return …
Basic Block: A group of consecutive instructions with a single entry point and a single exit point
76

Code Optimization Requirements

1. Preserve correctness
   - The speed of an incorrect program is irrelevant
2. On average, improve performance
   - Optimized code may be worse than original if unlucky
3. Be "worth the effort"
   - Is this example worth it?
     - 1 person-year of work to implement compiler optimization
     - 2 times increase in compilation time
     - 0.1% improvement in speed

77

Code Optimization

- Optimization improves programs by making them smaller or faster or both
  - They can be optimized for speed, memory usage, or program footprint
- Goal of code optimization is to translate a program into a new version that computes the same result more efficiently – by taking less time, memory, and other system resources
- Code optimization is achieved in two ways
  - Rearranging computations in a program to make them execute more efficiently
  - Eliminating redundancies in a program

78

Code Optimization

- Should not change the meaning of the program
  - Tries to improve the program, but the underlying algorithm is not affected
- Cannot replace an inefficient algorithm with an algorithm that is more efficient
- Also cannot fully utilize the instruction set of a particular architecture
  - Therefore, is independent of the target machine and the programming language

79

Optimizing Transformations

- Two types of optimizing transformations are
  - Local transformations
    - Applied over small segments of a program
    - Provides limited benefits at a low cost
      - Scope is a basic block which is a sequential set of instructions
  - Global transformations
    - Applied over larger segments consisting of loops or function bodies
    - Requires more analysis efforts to determine feasibility of an optimization
      - Control flow analysis and data flow analysis

80

Optimization and Performance

- How do optimizations improve performance?

Execution_time = num_instructions * CPI

(Cycles Per Instruction)

- Fewer cycles per instruction
  - Schedule instructions to avoid dependencies
  - Improve cache/memory behavior
    - E.g., locality
- Fewer instructions
  - E.g., target special/new instructions

81

Role of Optimizing Compilers

- Provide efficient mapping of program to machine
  - Eliminating minor inefficiencies
  - Code selection and ordering
  - Register allocation
- Don't (usually) improve asymptotic efficiency
  - Up to programmer to select best overall algorithm
  - Big-O savings are (often) more important than constant factors
    - But constant factors also matter

82

Limits of Optimizing Compilers

- Operate Under Fundamental Constraints
  - Must not cause any change in program behavior under any possible condition
- Most analysis is performed only within procedures
  - Inter-procedural analysis is too expensive in most cases
- Most analysis is based only on *static* information
  - Compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

83

Role of the Programmer

"How to write programs, given a good, optimizing compiler?"

- Don't smash code into oblivion
  - Hard to read, maintain, and assure correctness
- Do
  - Select best algorithm
  - Write code that's readable and maintainable
    - Procedures, recursion
    - Even though these factors can slow down code
  - Eliminate optimization blockers
    - Allows compiler to do its job
- Focus on Inner Loops
  - Do detailed optimizations where code will be executed repeatedly
  - Will get most performance gain here

84

Compiler Optimizations

- Machine independent (apply equally well to most CPUs)
  - Constant propagation
  - Constant folding
  - Common sub-expression elimination
  - Dead code elimination
  - Loop invariant code motion
  - Strength reduction
  - Function inlining
- Machine dependent (apply differently to different CPUs)
  - Instruction scheduling
  - Loop unrolling
  - Parallel unrolling
- Could do these manually, better if compiler does them
  - Many optimizations make code less readable/maintainable

85

Compiler Optimizations

- Constant Propagation (CP)
  - Replace variables with constants when possible

```
a = 5;                          CP
b = 3;
   ⋮
n = a + b;      n = 5 + 3
for (i = 0; i < n ; i++)
{
   ⋮
}
```

```
   ⋮                            CF
n = 5 + 3;      n = 8
for (i = 0; i < n ; i++)
{
   ⋮
}                               8
```

- Constant Folding (CF)
  - Evaluate expressions containing constants
  - Can lead to further optimization
    - E.g., another round of constant propagation

86

Compiler Optimizations

- Common Sub-expression Elimination (CSE)
  - Try to only compute a given expression once (assuming the variables have not been modified)

```
a = c * d;
    ⋮
d = (c * d + t) * u
```

⇒

```
a = c * d;
    ⋮
d = (a + t) * u
```

- Dead Code Elimination (DCE)
  - Compiler can determine if certain code will never execute

```
debug = 0;  // set to False
    ⋮
if (debug)
{
    ⋮
}
a = f(b);
```

⇒

```
debug = 0;
    ⋮
a = f(b);
```

87

Compiler Optimizations

- Loop Invariant Code Motion (LICM)
  - Loop invariant – value does not change across iterations
  - LICM will move invariant code out of the loop

```
for (i=0; i < 100 ;  ++i)
{
  for (j=0; j < 100 ; ++j)
  {
    for (k=0 ; k < 100 ; ++k)
    {
      a[i][j][k] = i*j*k;
    }
  }
}
```

⇒

```
for (i = 0; i < 100 ; ++i)
{
  t1 = a[i];
  for (j = 0; j < 100 ; ++j)
  {
    tmp = i * j;
    t2 = t1[j];
    for (k = 0 ; k < 100 ; ++k)
    {
      t2[k] = tmp * k;
    }
  }
}
```

Results in big performance savings as inner loop will execute 1M times!

88

Compiler Optimizations

- Strength Reduction
  - Replace expensive operation with simpler ones
  - Example: multiplication replaced by additions

```
y = x * 2;

a = b * 4;
```

⇒

```
y = x + x;

a = b + b + b + b;
```

Can be further optimized to: `a = b << 2`

89

Compiler Optimizations

- Function Inlining
  - Code size
    - Can decrease if small procedure body and few calls
    - Can increase if big procedure body and many calls
  - Performance
    - Eliminates call/return overhead
    - Can expose potential optimizations
    - Can be hard on instruction-cache if many copies made

```
int foo(int z)
{
  int m = 5;
  return z + m;
}
int main()
{
  …
  x = foo(x);
  …
}
```

⇒

```
int main()
{
  …
  {
    int foo_z = x;
    int m = 5;
    int foo_return = foo_z + m;
    x = foo_return;
  }
  …
}
```

⇒

```
int main()
{
  …
  x = x + 5;
  …
}
```

90

Compiler Optimizations

- Loop Unrolling
  - Reduces loop overhead
    - E.g., fewer adds to update j
    - Fewer loop condition tests
  - Enables more aggressive instruction scheduling
    - More instructions for scheduler to move around

```
j = 0;
while (j < 100)
{
   a[j] = b[j+1];
   j += 1;
}
```

⇒

```
j = 0;
while (j < 99)
{
   a[j] = b[j+1];
   a[j+1] = b[j+2];
   j += 2;
}
```

91

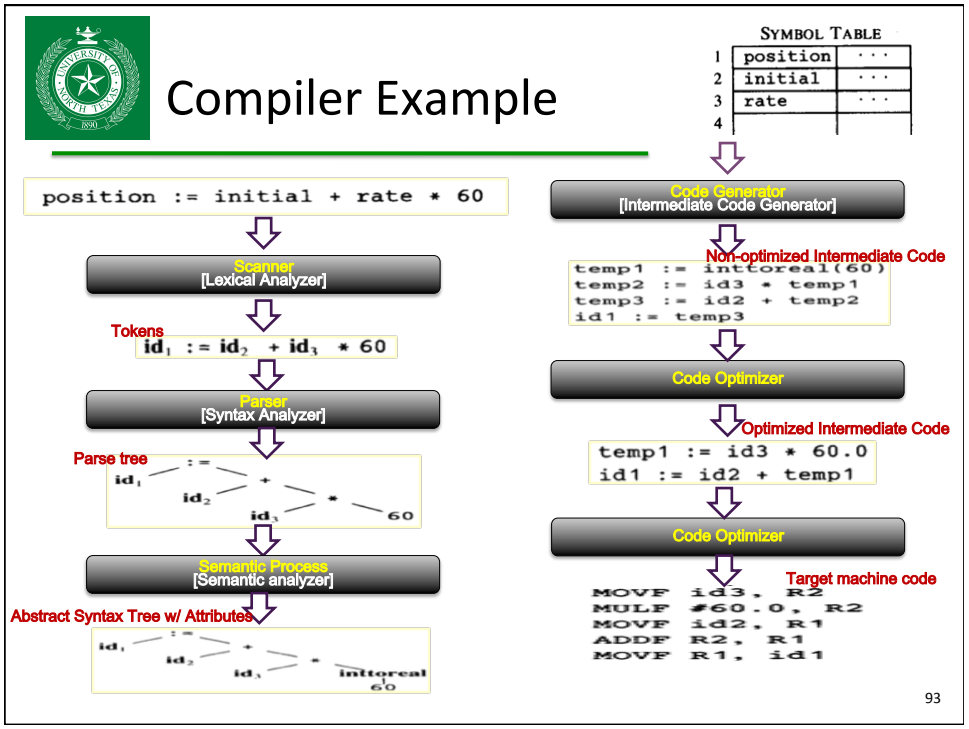
Optimizing Transformations

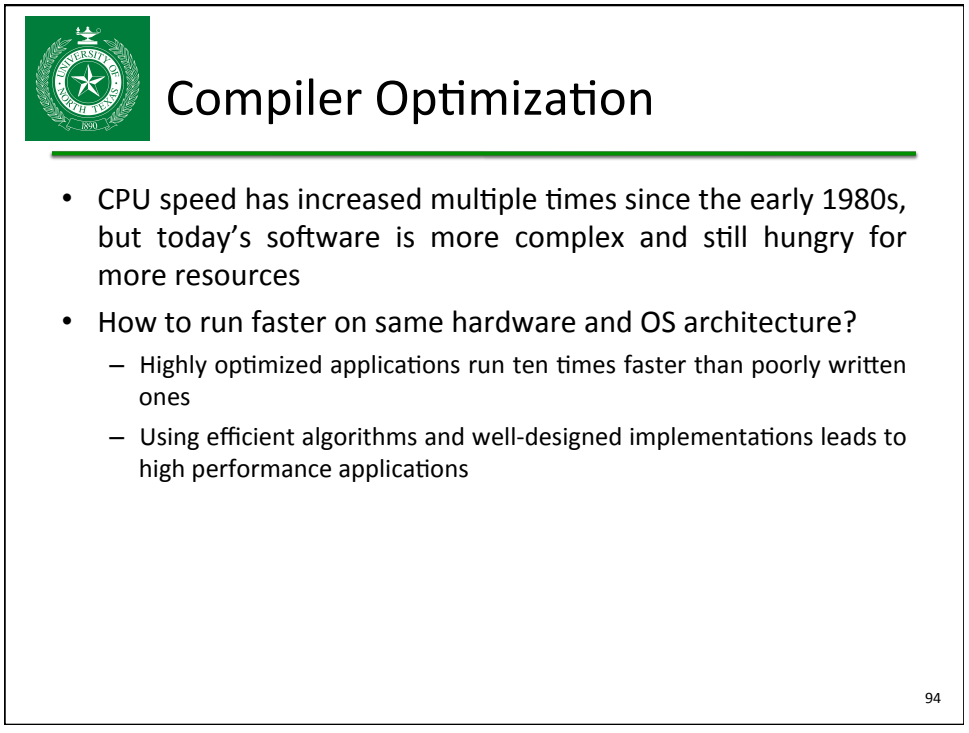
```
int test(int g, int h)
{
    int x = 4;
    int sum = 0;
    do
    {
        sum += f(g-1,h%4,g+(h%4));
        g = x * g + h;
    } while (g<128);
    return sum;
}
```

```
int test(int g, int h)
{
    int tmp = h%4;
    int sum = 0;
    do
    {
        sum += f(g-1,tmp,g+tmp);
        g = g<<2 + h;
    } while (g<128);
    return sum;
}
```

1. constant folding (x = 4)
2. dead code elimination (declaration of x)
3. common sub-expression elimination (h%4)
4. strength reduction (multiply → left shift)
5. loop invariant removal (h%4 moved outside of loop)

92


Compiler Example
position := initial + rate * 60
Scanner
[Lexical Analyzer]
Tokens
id1 := id2 + id3 * 60
Parser
[Syntax Analyzer]
Parse tree
Semantic Process
[Semantic analyzer]
Abstract Syntax Tree w/ Attributes
SYMBOL TABLE
1 position
2 initial
3 rate
4
Code Generator
[Intermediate Code Generator]
Non-optimized Intermediate Code
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
Code Optimizer
Optimized Intermediate Code
temp1 := id3 * 60.0
id1 := id2 + temp1
Code Optimizer
Target machine code
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
93


Compiler Optimization
• CPU speed has increased multiple times since the early 1980s, but today's software is more complex and still hungry for more resources
• How to run faster on same hardware and OS architecture?
– Highly optimized applications run ten times faster than poorly written ones
– Using efficient algorithms and well-designed implementations leads to high performance applications
94

Using Optimizing Compilers

- What does compiler optimization mean?
  - Execution time
  - Code size
  - Memory usage
  - Compile time
- Understanding and using all the features of an optimizing compiler is required for maximum performance with the least effort

95

gcc Compiler Optimization Levels

`-g`
- Include debug information, no optimization

`-O0`
- Default, no optimization

`-O1`
- Do optimizations that don't take too long
- CP, CF, CSE, DCE, LICM, inlining small functions

`-O2`
- Take longer optimizing, more aggressive scheduling

`-O3`
- Make space/speed trade-offs: loop unrolling, more inlining

`-Os`
- Optimize program size

96

Compiler Optimization

Option	Function
-Os	Signal that the generated code should be optimised for code size. The compiler will not care about the execution performance of the generated code.
-O0	No optimisation. This is the default. GCC will generate code that is easy to debug but slower and larger than with the incremental optimization levels outlined below.
-O1 or -O	This will optimise the code for both speed and size. Most statements will be executed in the same order as in the C/C++ code and most variables can be found in the generated code. This makes the code quite suitable for debugging.
-O2	Turn on most optimizations in GCC except for some optimisations that might drastically increase code size. This also enables instruction scheduling, which allows instructions to be shuffled around to minimize CPU stall cycles because of data hazards and dependencies, for CPU architectures that might benefit from this. Overall this option makes the code quite small and fast, but hard to debug.
-O3	Turn on some extra performance optimisations that might drastically increase code size but increase performance compared to the -O2 and -O1 optimization levels. This includes performing function inlining

```
gcc helloworld.c -Q -O3 --help=optimizers > /tmp/03-opts
gcc helloworld.c -Q -O2 --help=optimizers > /tmp/02-opts
diff /tmp/02-opts /tmp/03-opts | grep enabled
```

97