Compiler design

Overview

- Compiler Front-End
  - What is a compiler?
  - Lexical Analysis
  - Syntax Analysis
  - Parsing
- Compiler Back-End
  - Code Generation
  - Register Allocation
  - Optimization
- Specific Examples
  - lex
  - yacc
  - lcc

What is a Compiler?

Example of tasks of compiler

1. Add two numbers
2. Move numbers from one location to another
3. Move information between CPU and memory

Software Translator

Lexical Analysis

First phase of compiler

isolate words/tokens

Example of tokens:

- key words – while, procedure, var, for,..
- identifier – declared by the programmer
- Operators – +, -, *, /, <>, ...
- Numeric – numbers such as 124, 12.35, 0.09E-23, etc.
- Character constants
- Special characters
- Comments

Syntax Analysis

- **What is Syntax Analysis?**

  Second phase of the compiler

  Also called Parser

- **What is the Parsing Problem?**

- **How is the Parsing problem solved?**

  Top-down and Bottom-up algorithm

Top-Down Parsing

What does it do?

One Method: Pushdown Machine

Example:

Consider the simple grammar:

1. $S \rightarrow 0\ S\ 1\ A$
2. $S \rightarrow 1\ 0\ A$
3. $A \rightarrow 0\ S\ 0$
4. $A \rightarrow 1$

Example

Process to construct a Pushdown Machine

1. Build a table with each column labeled by a terminal symbol (and endmarker ↵) and each row labeled by a nonterminal or terminal symbol (and bottom marker ∇)
2. For each grammar rule of the form A $\rightarrow$ *a*α, fill in the cell in row A and column a with with: REP(αr*a*), *retain*, where αr represents α reversed
3. Fill in the cell in row *a* and column *a* with pop, *advance*, for each terminal symbol *a*.
4. Fill in the cell in row ∇ and column ↵ with *Accept*.
5. Fill in all other cells with *Reject*.
6. Initialize the stack with ∇ and the starting terminal.

Bottom-Up Parsing

What does it do?

Two Basic Operations:

1. Shift Operation
2. Reduce Risk Operation

Why Split the Compiler

- Front- End is Machine Independent
- Front-End can be written in a high level language
- Re-use Oriented Programming
- Back-End is Machine Dependent
- Lessens Time Required to Generate New Compilers
- Makes developing new programming languages simpler

Code Generation

- Convert functions into simple instructions
  - Simple
  - Complex
- Addressing the operands
  - Base Register
  - Offset
  - Examples

Single Pass vs. Multiple pass

- **Single pass**
  - Creates a table of Jump Instructions
  - Forward Jump Locations are generated incompletely
  - Jump Addresses entered into a fix-up table along with the label they are jumping to
  - As label destinations encountered, it is entered into the table of labels
  - After all inputs are read, CG revisits all of these problematic jump instructions

- **Multiple pass**
  - No Fix-Up table
  - In the first pass through the inputs, CG does nothing but generate table of labels.
  - Since all labels are now defined, whenever a jump is encountered, all labels already have pre-defined memory location.
  - Possible problem: In first pass, CG needs to know how many MLI correspond to a label.
  - Major Drawback-Speed

Register Allocation

- Assign specific CPU registers for specific values
- CG must maintain information on which registers:
  - Are used for which purposes
  - Are available for reuse
- Main objective:
  - Maximize the utilization of the CPU registers
  - Minimize references to memory locations
- Possible uses for CPU registers
  - Values used many times in a program
  - Values that are computationally expensive
- Importance?
  - Efficiency
  - Speed

An Example

Example - For the following 2 statement program segment, determine a smart register allocation scheme:

$$A = B + C * D$$
$$B = A - C * D$$

Simple Register Allocation
LOD (R1,C)
MUL (R1,D)
STO (R1,Temp)
LOD (R1,B)
ADD (R1,Temp)
STO (R1,A)
LOD (R1,C)
MUL (R1,D)
STO (R1,Temp2)
LOD (R1,A)
SUB (R1,Temp2)
STO (R1,B)
Net Result
12 instructions and memory ref.

Smart Register Allocation	
LOD (R1,C)	
MUL (R1,D)	**C*D**
LOD (R2,B)	
ADD (R2,R1)	**B+C*D**
STO (R2,A)	
SUB (R2,R1)	**A-C*D**
STO (R2,B)	
Net Result	
7 instruc. And 5 mem. refs.	

Register Allocation Algorithm

- RAA determines how many registers will be needed to evaluate an expression.

- Determines the Sequence in which sub-expressions should be evaluated to minimize register use

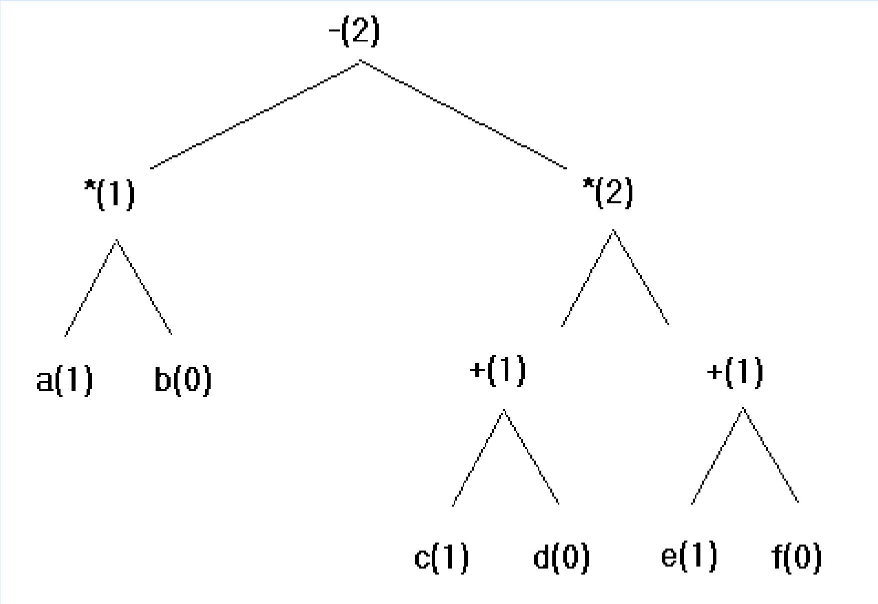
How does RAA work?

- Construct a tree starting at the bottom nodes
- Assign each leaf node a weight of:
  - 1 if it is the left child
  - 0 is it is the right child
- The weight of each parent node will be computed by the weights of the 2 children as follows:
  - If the 2 children have different weights, take the max.
  - If the weights are the same, the parent's weight is w+1
- The number of CPU registers is determined by the highest summed weight at any stage in the tree.

Example of RAA

Example - For the following 2 statement program segment, determine a smart register allocation scheme:

*A*B – (C+D) * (E+F)*



LOD (R1,c)			
ADD (R1,d)	R1 = c +d		
LOD (R2,e)			
ADD (R2,f)	R2 = e + f		
MUL (R1,R2)	R1 = (c + d) * (e + f)		
LOD (R2,a)			
MUL (R2,b)	R2 = a * b		
SUB (R2,R1)	R2 = a * b - (c + d) * (e + f)		

Optimization

- **Global**
  - Directed Acyclic Graphs (DAGs)
  - Data Flow Analysis
  - Moving Loop Invariant Code
  - Other Mathematical Transformations
- **Local**
  - Load Store Optimization
  - Jump over Jump Optimization
  - Simple Algebraic Optimization

Main Problem with optimization techniques: Debugging is more difficult

Analysis of specific compilers

Programs to be discussed:

- lex – Programming utility that generates a lexical analyzer
- yacc – Parser generator
- lcc - ANSI C compiler

Platforms:

- All three programs designed for use on Unix
- lcc runs under DOS and Unix

lex Programming Utility

General Information:

- Input is stored in a file with *.l extension
- File consists of three main sections
- lex generates C function stored in lex.yy.c

Using lex:

1) Specify words to be used as tokens (Extension of regular expressions)
2) Run the lex utility on the source file to generate **yylex( )**, a C function
3) Declares global variables char* yytext and int yyleng

lex Programming Utility

Three sections of a lex input file:

```
/* C declarations and #includes lex definitions */
%{ #include "header.c"
int i; }%

%%
/* lex patterns and actions */
{INT}           {sscanf (yytext, "%d", &i);
                 printf("INTEGER\n");}

%%
/* C functions called by the above actions */
{ yylex(): }
```

yacc Parser Generator

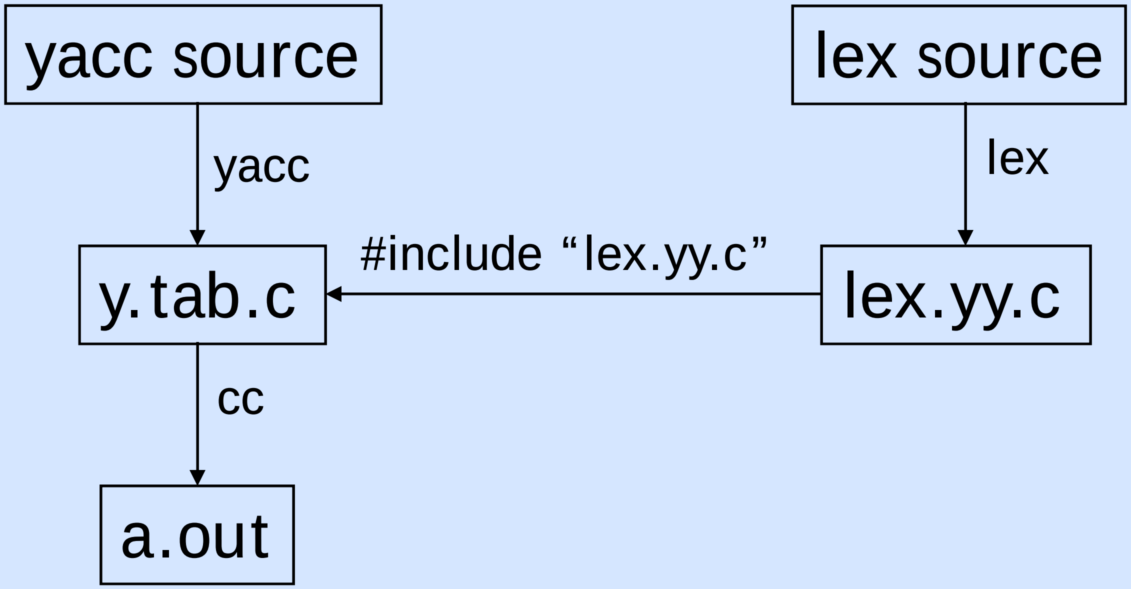
General Information:

- Input is specification of a language
- Output is a compiler for that language
- yacc generates C function stored in y.tab.c
- Public domain version available **bison**

Using yacc:

1) Generates a C function called **yyparse()**
2) **yyparse()** may include calls to yylex()
3) Compile this function to obtain the compiler

yacc Parser Generator



- Input source file – similar to lex input file
- Declarations, Rules, Support routines
- Four parts of output atom:

(Operation, Left Operand, Right Operand, Result)

lcc Compiler

General Information:

- Retargetable ANSI C compiler (machine specific parts that are easy to replace)
- Different stages of code:
  1. Preprocessed code
  2. Tokens
  3. Trees
  4. DAG (directed acyclic graphs)
  5. Assembly language

Test program:

```
int round(f) float f; {
        return f+0.5;   /* truncates the variable f */
 }
```

lcc Compiler

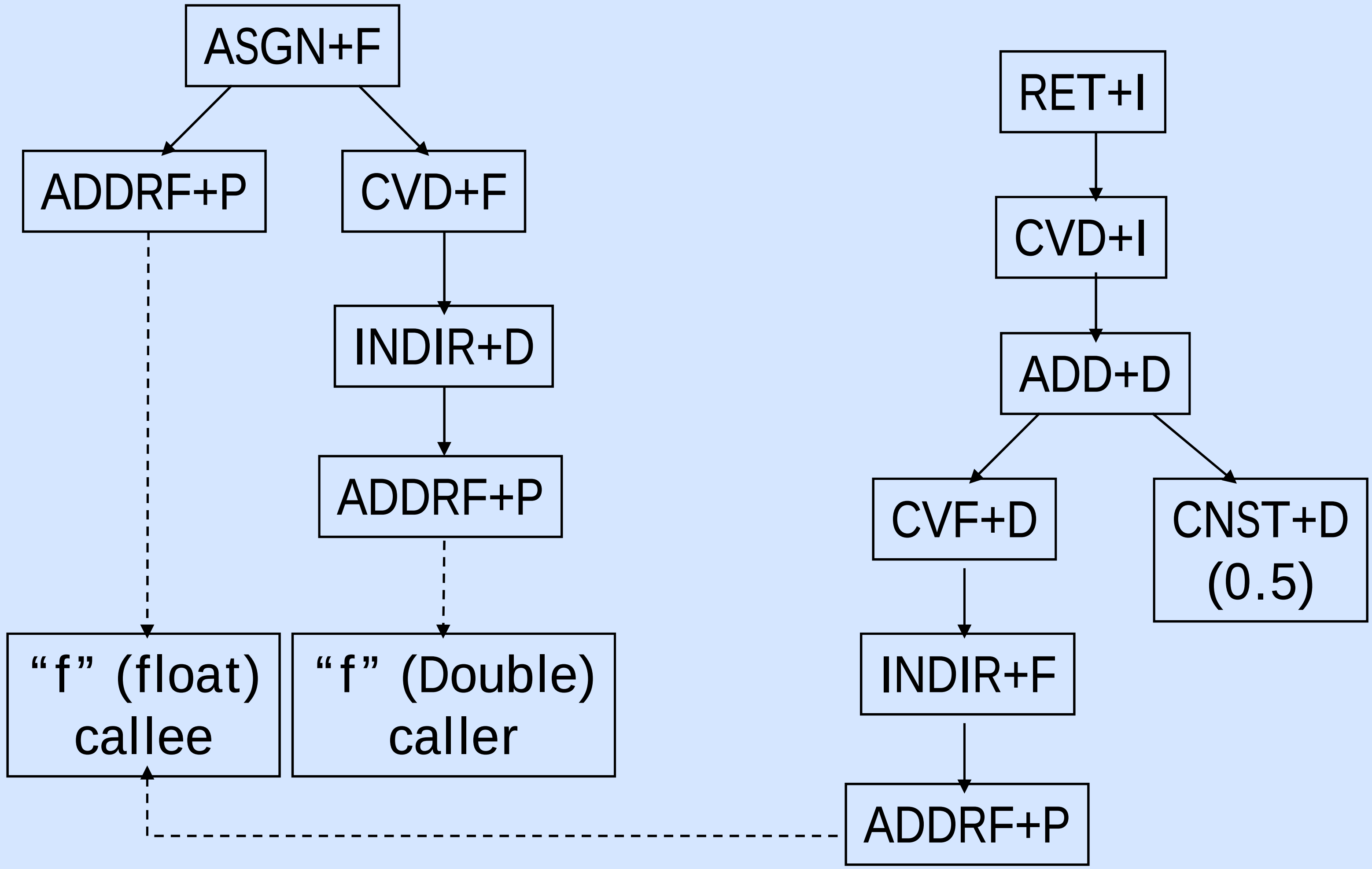
Token Stream

Tokens	Values
INT	inttype
ID	“round”
‘(‘	
ID	“f”
‘)’	
FLOAT	floattype
ID	“f”
‘;’	

Tokens	Values
‘{‘	
RETURN	
ID	“f”
‘+’	
FCON	0.5
‘;’	
‘}’	
EOI	

lcc Compiler

Syntax Trees



lcc Compiler

Register	Assembler Template
	fld qword ptr %a[ebp] \n
	fstp dword ptr %a[ebp] \n
	fld dword ptr %a[ebp] \n
	#nop \n
	fadd qword ptr %a \n
	sub esp, 4 \n
	fistp dword ptr 0[esp] \n
eax	pop %c \n
	#ret \n

Conclusion

- Compiler Front-End
- Compiler Back-End
- Specific Examples