Compilers & Translator Writing Systems

Prof. R. Eigenmann

ECE573, Fall 2005

http://www.ece.purdue.edu/~eigenman/ECE573

Compilers are Translators

- Fortran
- C
- C++
- Java
- Text processing language
- Command Language
- Natural language

translate →

- Machine code
- Virtual machine code
- Transformed source code
- Augmented source code
- Low-level commands
- Semantic components

Compilers are Increasingly Important

Specification languages
↑
High-level languages
↑
Assembly languages

Increasingly high level **user interfaces** for specifying a computer problem/solution

↑

The compiler is the translator between these two diverging ends

↓

Increasingly complex **machines**

Non-pipelined processors
Pipelined processors
Speculative processors
Worldwide "Grid"

ECE573, Fall 2005 3

Assembly code and Assemblers

Compiler → assembly code → Assembler → machine code

- Assemblers are often used at the compiler back-end.
- Assemblers are low-level translators.
- They are machine-specific,
- and perform mostly 1:1 translation between mnemonics and machine code, except:
  - symbolic names for storage locations
    - program locations (branch, subroutine calls)
    - variable names
  - macros

ECE573, Fall 2005 4

Interpreters

- "Execute" the source language directly.
- Interpreters directly produce the result of a computation, whereas compilers produce executable code that can produce this result.
- Each language construct executes by invoking a subroutine of the interpreter, rather than a machine instruction.

Examples of interpreters?

Properties of Interpreters

- "execution" is immediate
- elaborate error checking is possible
- bookkeeping is possible. E.g. for garbage collection
- can change program on-the-fly. E.g., switch libraries, dynamic change of data types
- machine independence. E.g., Java byte code

BUT:

- is slow; space overhead

Job Description of a Compiler

At a very high level a compiler performs two steps:

1. analyze the source program
2. synthesize the target code

ECE573, Fall 2005 7

Block Diagram of a Compiler

compiler passes:

Scanner — Tokenizer, lexer, also processes comments and directives. Token description via regular expressions → **scanner generators**. Takes non-trivial time.

↓

Parser — Grouping of tokens. CFG (context free grammar). Error detection and recovery. **Parser generator** tools.

↓

Semantic Routines — The heart of a compiler. Deals with the meaning of the language constructs. Translation to IR. Abstract code generation. **Not automatable**, but can be formalized through Attribute Grammars.

↓

Optimizer — Generate functionally equivalent but improved code. Complex. Slow. User options to set level. Peephole vs. global optimization. Source vs. object code optimization. Usually hand-coded. **Automation is a research topic**, e.g. template optimizers.

↓

Code Generator — Machine-specific, although similarities for classes of machines. Instruction selection, register allocation, instruction scheduling.

ECE573, Fall 2005 8

Compiler Writing Tools

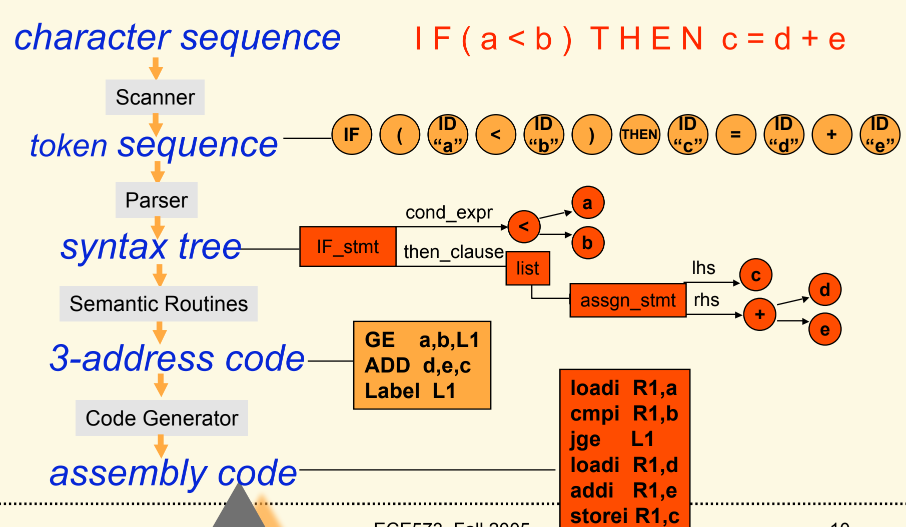
Other terms: compiler generators, compiler compilers

- scanner generators, example: lex
- parser generators, example: yacc
- symbol table routines,
- code generation aids,
- (optimizer generators, still a research topic)

These tools are useful, but bulk of work for compiler writer is in semantic routines and optimizations.

ECE573, Fall 2005 9

Compiler Input, Output and Intermediate Representations



ECE573, Fall 2005 10

Symbol and Attribute Tables

- Keep information about identifiers: variables, procedures, labels, etc.
- The symbol table is used by most compiler passes
  - Symbol information is entered at declaration points,
  - Checked and/or updated where the identifiers are used in the source code.

```
Program Example
Integer ii;
...
ii = 3.5;
...
print ii;
```

Symbol Table

Name	Type	Scope
ii	int	global
...		

ECE573, Fall 2005 11

Sequence of Compiler Passes

In general, all compiler passes are run in sequence.

- They read the internal program representation,
- process the information, and
- generate the output representation.

For a simple compiler, we can make a few simplifications. For example:

- Semantic routines and code generator are combined
- There is no optimizer
- All passes may be combined into one. That is, the compiler performs all steps in one run.
  - One-pass compilers do not need an internal representation. They process a syntactic unit at a time, performing all steps from scanning to code generation.

Example: (simple) Pascal compilers

ECE573, Fall 2005 12

Language Syntax and Semantics

An important distinction:

- Syntax defines the structure of a language.

  E.g., an IF clause has the structure:
  IF (*expression*) THEN *statements*

- Semantics defines its meaning.

  E.g., an IF clause means:
  test the *expression*; if it evaluates to true, execute the *statements*.

ECE573, Fall 2005 13

Context-free and Context-sensitive Syntax

- The context-free syntax part specifies legal sequences of symbols, independent of their type and scope.
- The context-sensitive syntax part defines restrictions imposed by type and scope.
  - Also called the "static semantics". E.g., all identifiers must be declared, operands must be type compatible, correct #parameters.
  - Can be specified informally or through *attribute grammar.*

ECE573, Fall 2005 14

Example

context-free and context-sensitive syntax parts

- CFG:
  E1 → E2 + T

"The term E1 is composed of E2, a "+", and a T"

- Context-sensitive part, specified through Attribute Grammars

"Both E1 and T must be of type numeric"

(E2.type=numeric) and (T.type=numeric)

ECE573, Fall 2005 15

(Execution) Semantics

(a.k.a. runtime semantics)

- Often specified informally
- Attempts to formalize execution semantics (have not reached practical use):
  - Operational or interpreter model: (state-transition model). E.g., Vienna definition language, used for PL/1. Large and verbose.
  - Axiomatic definitions: specifies the effect of statements on variable relationships. More abstract than operational model.

ECE573, Fall 2005 16

Execution Semantics

- Denotational Semantics:
  - More mathematical than operational semantics. Includes the notion of a “state”.
  - For example, the semantics of E[T1+T2] :

    If E[T1] is integer and E[T2] is integer
    then the result is (E[T1]+E[T2]) else error
  - Is an important research area.
    Goal: compiler generators from D.S.

ECE573, Fall 2005 17

Significance of Semantics Specification

- Leads to a well-defined language, that is complete and unambiguous.
- Automatic generation of semantics routines becomes possible.

Note: A compiler is a de-facto language definition. (what’s not fully defined in the language specs is defined in the compiler implementation)

ECE573, Fall 2005 18

Compiler and Language Design

There is a strong mutual influence:

- hard to compile languages are hard to read
- easy to compile language lead to quality compilers, better code, smaller compiler, more reliable, cheaper, wider use, better diagnostics.

Example: dynamic typing

- seems convenient because type declaration is not needed

However, such languages are

- hard to read because the type of an identifier is not known
- hard to compile because the compiler cannot make assumptions about the identifier's type.

ECE573, Fall 2005 19

Compiler and Architecture Design

- Complex instructions were available when programming at assembly level.
- RISC architecture became popular with the advent of high-level languages.
- Today, the development of new instruction set architectures (ISA) is heavily influenced by available compiler technology.

ECE573, Fall 2005 20

So far we have discussed ...

Structure and Terminology of Compilers

- Tasks of compilers, interpreters, assemblers
- Compiler passes and intermediate representations
- Scope of compiler writing tools
- Terminology: Syntax, semantics, context-free grammar, context-sensitive parts, static semantics, runtime/execution semantics
- Specification methods for language semantics
- Compiler, language and architecture design

Next: An example compiler

ECE573, Fall 2005 21

The Micro Compiler

An example of a one-pass compiler for a mini language

22

The Micro Language

- integer data type only
- implicit identifier declaration. 32 chars max. [A-Z][A-Z0-9]*
- literals (numbers): [0-9]*
- comment: -- non-program text <end-of-line>
- Program :

  BEGIN Statement, Statement, ... END

ECE573, Fall 2005 23

Micro Language

- Statement:
  - Assignment:

    ID := Expression

    Expression can contain infix + -, ( ) , Ids, Literals
  - Input/Output:

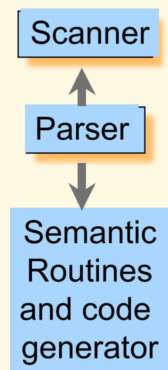
    READ(ID, ID, …)

    WRITE(Expression, Expression, …)

ECE573, Fall 2005 24

Implementation of the Micro Compiler

- 1-pass compiler. No explicit intermediate representation.
- Scanner: tokenizes input character stream. Is called by parser on-demand.
- Parser recognizes syntactic structure, calls Semantic Routines.
- Semantic routines, in turn, call code generation routines directly, producing code for a 3-address virtual machine.
- Symbol table is used by Semantic routines only



Scanner for Micro

Interface to parser: token scanner();

```
typedef enum token_types {
  Begin, End, Read, Write, ID, Intliteral,
  Lparem, Rparen, Semicolon, Comma, Assignop,
  Plusop, Minusop, ScanEof} token;
```

Scanner Algorithm: (see textbook p. 28/29)

Scanner Operation

- scanner routine:
  - identifies the next token in the input character stream :
    - read a token
    - identify its type
    - return token type and "value"

Scanner Operation (2)

- Skip spaces.
- If the first non-space character is a
  - letter: read until non-alphanumeric. Put in buffer. Check for reserved words. Return reserved word or identifier.
  - digit: read until non-digit. Put in buffer. Return number (INTLITERAL).
  - ( ) ; , + → return single-character symbol.
  - : : next must be = → return ASSIGNOP.
  - \- : if next is also - → comment. Skip to EOL.
    Read another token.
    Otherwise return MINUSOP.
- "unget" the next character that had to be read for Ids, reserved words, numbers, and minusop.

Note: Read-ahead by one character is necessary.

Grammar and Parsers

- Context-Free Grammar (CFG) is most often used to specify language syntax.
- (Extended) Backus-Naur form is a convenient notation.
- It includes a set or rewriting rules or *Productions*,

  A production tells us how to compose a *non-terminal* from *terminals* and other non-terminals.

ECE573, Fall 2005 29

Micro Grammar

```
Program         ::= BEGIN Statement-list END
Statement-list  ::= Statement {Statement}
Statement       ::= ID := Expression ;  |
                    READ ( Id-list ) ;  |
                    WRITE ( Expr-list ) ;
Id-list         ::=  ID {, ID }
Expr-list       ::= Expression {, Expression}
Expression      ::= Primary { Add-op Primary }
Primary         ::= ( Expression )   |
                    ID               |
                    INTLITERAL
Add-op          ::= PLUSOP  | MINUSOP
System-goal     ::= Program SCANEOF
```

ECE573, Fall 2005 30

Given a CFG, how do we parse a program?

Overall operation:

- start at goal term, rewrite productions (from left to right)
  - if it's a terminal: check if it matches an input token,
  - else (it's a non-terminal):
    - if there is a single choice for a production: take this production,
    - else: take the production that matches the first token.
- if the expected token is not there, that means syntax error.

Notes:

•1-token lookahead is necessary.

•Static semantics is not checked (for Micro).

ECE573, Fall 2005 31

Operator Precedence

- Operator precedence is also specified in the CFG ⇒ CFG tells what is legal syntax **and** how it is parsed.

For example,

```
Expr ::= Factor { + Factor }
Factor ::= Primary { * Primary }
Primary ::= ( Expr ) | ID | INTLITERAL
```

specifies the usual precedence rules: * before **+**

ECE573, Fall 2005 32

Recursive Descent Parsing

Each production P has an associated procedure, usually named after the *nonterminal* on the LHS.

Algorithm for P():

- for *nonterminal* A on the RHS : call A().
- for *terminal* t on the RHS : call match(t), (matching the token t from the scanner).
- if there is a choice for B: look at First(B)

  First(B) is the set of terminals that B can start with. (this choice is unique for LL(1) grammars). Empty productions are used only if no other choice.

An Example Parse Procedure

Program ::= **BEGIN** Statement-list **END**

```
Procedure Program()
  match(Begin);
  StatementList();
  match(End);
END
```

Another Example Parse Procedure

Id-list ::= **ID** { **,** **ID** }

```
Procedure IdList()
  match(ID);
  WHILE LookAhead(Comma)  match(ID);
END
```


Parser Code for Micro

(text pages 36 - 38)

Things to note:

- there is one procedure for each nonterminal.
- nonterminals with choices have **case** or **if** statements.
- an optional list is parsed with a loop construct, testing the First() set of the list item.
- error handling is minimal.

Semantic Processing and Code Generation

- Micro will generate code for a 3-address machine: OP A,B,C performs A *op* B → C
- Temporary variables may be needed to convert expressions into 3-address form. Naming scheme: **Temp&1, Temp&2, ...**

D=A+B*C ⟶

```
MULT  B,C,TEMP&1
ADD   A,Temp&1,D
```

ECE573, Fall 2005 37

Semantics Action Routines and Semantic Records

- How can we facilitate the creation of the semantic routines?
- Idea: call routines that generate 3-address code at the right points during parsing.

These *action routines* will do one of two things:

1. Collect information about parsed symbols for use by other action routines. The information is stored in ***semantic records***.
2. Generate the code using information from semantic records and the current parse procedure.

ECE573, Fall 2005 38

Semantics Annotations

- Annotations are inserted in the grammar, specifying when semantics routines are to be called.

as_stmt → ID = expr #asstmt
expr → term + term #addop
term → ident #id | number #num

- Consider A = B + 2
  - num() and id() write *semantic records* of ID names and number values.
  - addop() generates code for the *expr* production, using information from the semantic records created by num() and id().
  - asstmt() generates code for the assignment to A, using the result of B+2 generated by addop()

ECE573, Fall 2005 39

Annotated Micro Grammar

Program ::= #start **BEGIN** Statement-list **END**
Statement-list ::= Statement {Statement}
Statement ::= **ID :=** Expression**;** #assign |
READ (Id-list **) ; |**
WRITE (Expr-list **) ;**
Id-list ::= Ident #read_id {**,** Ident #read_id }
Expr-list ::= Expression #write_expr {**,** Expression #write_expr }
Expression ::= Primary { Add-op Primary #gen_infix}
Primary ::= **(** Expression **)** |
Ident |
INTLITERAL #process_literal
Ident ::= **ID** #process_id
Add-op ::= **PLUSOP #**process_op **|**
MINUSOP #process_op
System-goal ::= Program **SCANEOF #**finish

ECE573, Fall 2005 40

Semantics Action Routines for Micro

- (text, pages 41 - 45)
- A procedure corresponds to each annotation of the grammar.
- The parsing routines have been extended to return information about the identified constructs. E.g.,

void expression(expr_rec *results)

ECE573, Fall 2005 41

So far we have covered ...

- Structure of compilers and terminology
- Scanner, parser, semantic routines and code generation for a one-pass compiler for the Micro language

Next: Scanning

ECE573, Fall 2005 42

Scanning

regular expressions

finite automata

scanner generators

practical considerations

43

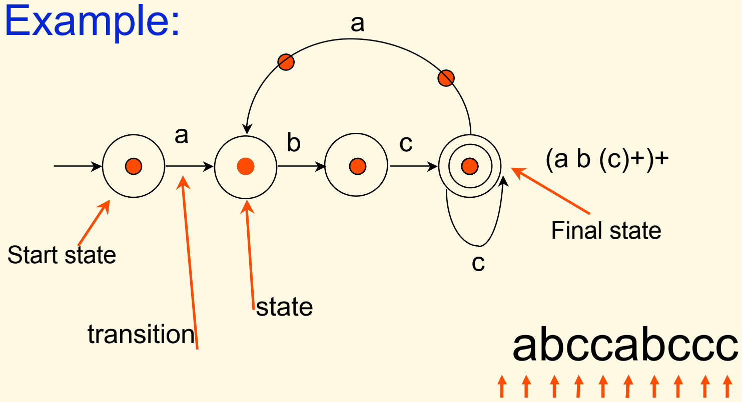
Regular Expressions

- Examples of regular expressions:
  - D=(0|…|9)   L=(A|…|Z)
  - comment = -- Not(Eol)*Eol
  - Literal = D+.D+
  - ID = L(L|D)*(_(L|D)+)*
  - comment2 = ##((#|λ)Not(#))*##
- regular sets = strings defined by reg. exp.
- λ = empty string,
- * = 0 or more repetitions, + = 1 or more repetitions

ECE573, Fall 2005   44

Finite Automata

- Example:



ECE573, Fall 2005 45

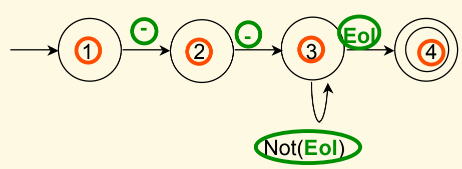
Transition Tables

- unique transitions => FA is deterministic (DFA)
- DFAs can be represented in transition tables
- T[s][c] indicates the next state after state *s*, when reading character *c*

Consider: -- Not(Eol)* Eol

Example:
--b

State	Character				
	-	Eol	a	b	....
1	2				
2	3				
3	3	4	3	3	3
4					



ECE573, Fall 2005 46

Finite Automata Program

Given a transition table, we can easily write a program that performs the scanning operation:

```
state = initial_state;
while (TRUE) {
    next_state = T[state][current_char];
    if (nextstate==ERROR) break;
    state=next_state;
    if (current_char==EOF) break;
    current_char = getchar();
}
if (is_final_state(state))
  //a valid token is recognized
else
  lexical_error(current_char);
```

ECE573, Fall 2005 47

Same program “conventionally”

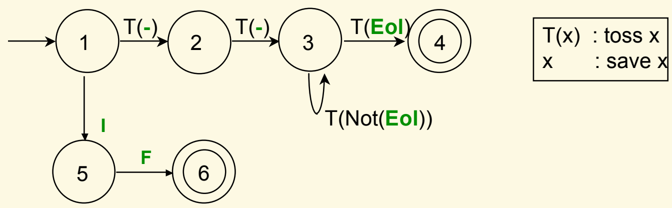
The previous program looks different from the scanner shown on textbook pages 28/29. We could write the scanner in that way too:

```
if (current_char == ‘-’) {
   current_char = getchar();
   if (current_char == ‘-’) {
     do
       current_char=get_char(); // skip character
     while (current_char != ‘\n’) ;
   } else {
     ungetc(current_char);
     lexical_error(current_char);
   }
}
else lexical_error(current_char);
// a valid token is recognized
```

ECE573, Fall 2005 48

Transducer

- A simple extension of a FA, which also outputs the recognized string.
- Recognized characters are output, “the rest” is discarded.



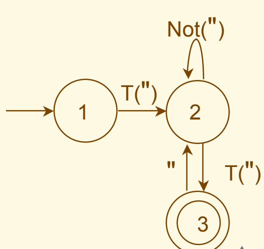
We need this for tokens that have a *value*.

ECE573, Fall 2005 49

Example: A FA with Transducer for quoted Strings

Quoted string, double quotes within string

(" (Not(") | " ")* ")



Examples:

"EE468" → EE468
"it's an ""easy"" job" → it's an “easy” job
"""Polaris"" beats ""SUIF""" → “Polaris” beats “SUIF”

ECE573, Fall 2005 50

Scanner Generators

- We will discuss ScanGen, a scanner generator that produces tables for a finite automata driver program, and
- Lex, which generates a scanner procedure directly, making use of user-written "filter" procedures.

ECE573, Fall 2005 51

Scan Gen

User defines the input to ScanGen in the form of a file with three sections:

- Options,
- Character Classes,
- Token Definitions:

Token *name* **{***minor*,*major***}** **=** *regular expression*

- Regular expression can include except clauses, and
- {Toss} attributes

Example of ScanGen input:

textbook page 61: extended Micro

ECE573, Fall 2005 52

ScanGen Driver

- The driver routine provides the actual scanner routine, which is called by the parser.

```
void scanner(codes *major,
             codes *minor,
             char *token_text)
```

- It reads the input character stream, and drives the finite automata, using the tables generated by ScanGen, and returns the found token.

ECE573, Fall 2005 53

ScanGen Tables

- The finite automata table has the form
  next_state[NUMSTATES][NUMCHARS]
- In addition, an action table tells the driver when a complete token is recognized and what to do with the "lookahead" character:

  action[NUMSTATES][NUMCHARS]

ECE573, Fall 2005 54

Action Table

- The action table has 6 possible values:

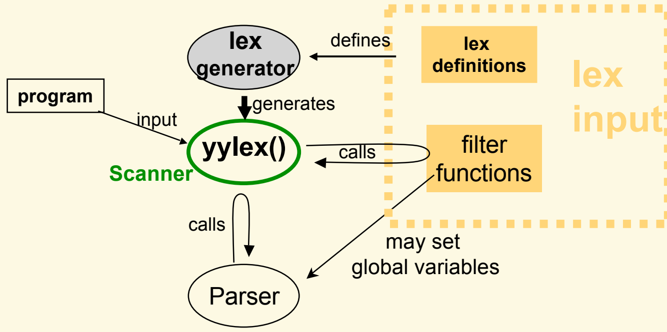
ERROR	scan error.
MOVEAPPEND	current_token += ch and go on.
MOVENOAPPEND	discard ch and go on.
HALTAPPEND	current_token += ch, token found, return it.
HALTNOAPPEND	discard ch, token found, return it.
HALTREUSE	save ch for later reuse, token found, return it.

Driver program on textbook pages 65,66

Lex

- Best-known scanner generator under UNIX.
- Has character classes and regular expressions similar to ScanGen.
- Calls a user-defined “filter” function after a token has been recognized. This function preprocesses the identified token before it gets passed to the parser.
- No {Toss} is provided. Filter functions take this role.
- No exceptions provided. But several regular expressions can match a token. Takes the first one.

Lex Operation



Example of a Lex input see textbook Page 67 (extended Micro language)

Practical Scanner Considerations: Handling Reserved Words

- Keywords can be written as regular expressions. However, this would lead to a significant increase in FA size.
- Special lookup as “exceptions” is simpler.

Exercise: Extend Regular Expressions for Micro so that keywords are no exceptions.

Practical Considerations: Additional Scanner Functions

- handling compiler directives

```
C$ PARALLEL
  DO I=1,10
    A(I)=B(I)
  ENDDO
```

Simple directives can be parsed in the scanner

- Include files and conditional compilation
  - minimal parsing is necessary to understand these directives as well

ECE573, Fall 2005 59

Practical Considerations: Pretty printing of source file

issues:

- include error messages (also, handle delayed error messages) and comment lines
- edit lines to include line numbers, pretty print, or expand macros
- deal with very long lines

→ keep enough position information and print at end

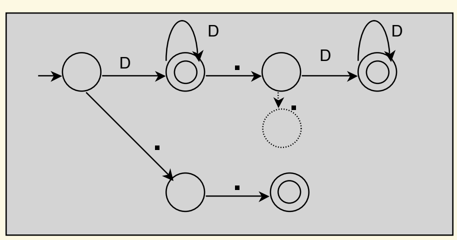
ECE573, Fall 2005 60

Practical Considerations: Generating symbol table entries

- in simple languages the scanner can build the symbol table directly
- this does not work where variable scopes need to be understood. In this case the parser will build the symbol table.

Multi-Character Lookahead

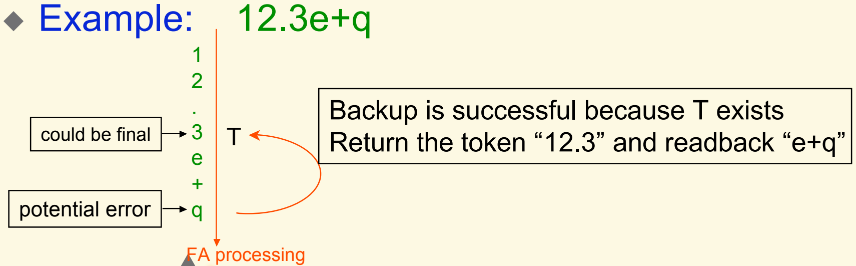
- Fortran: DO I=1,100 DO I=1.100
- Pascal: 23.85 23..85



2 Solutions: Backup and “Special Action” State

General Scheme for Multi-Character Lookahead

- remember states (T) that can be final states
- buffer the characters from then on
- if stuck in non-final state, backup to T.
- Example: 12.3e+q



ECE573, Fall 2005 63

Lexical Error Recovery

- what to do on lexical error?
  - 1. Delete characters read so far. Restart scanner.
  - 2. Delete the first character read. Restart scanner.
    - This would not work well for runaway strings. Possible solution: runaway string token. Warning if a comment contains the beginning of another comment.

ECE573, Fall 2005 64

Translating Regular Expressions into Finite Automata

- Regular Expression can be composed of:

a	character “a”
λ	empty expression
A | B	expression A or expression B
AB	A followed by B
A*	A repeated 0 or more times

Mini Exercise: how can A+ be built?

ECE573, Fall 2005 65

Building the FA

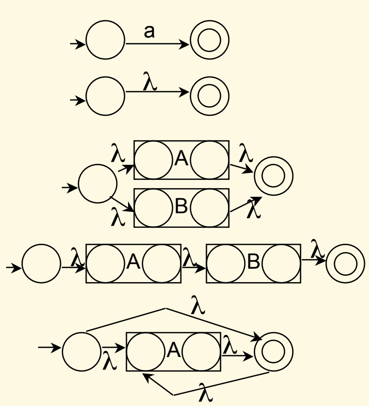
a

λ

A | B

AB

A*



Creating such automata results in non-deterministic FAs. (several transitions are possible)

ECE573, Fall 2005 66

Building DFAs from NFAs

- The basic idea for building a deterministic FA from a non-deterministic FA is to group nodes that can be reached via the same character into one node.
- Algorithm see textbook p. 82

Optimizing FA

- The built FA are not necessarily minimal.
- The basic idea of the optimization algorithm is like this:
  - 1. start with two big nodes, the first includes all final states, the second includes all other nodes.
  - 2. successively split those nodes whose transitions lead to different nodes.

Algorithm see textbook page 85

So far we have covered ...

- Compiler overview. Quick tour through the major compiler passes.
- Scanners: Finite automata, transition tables, regular expressions, scanner generation methods and algorithms.

Next:

- Parsers

ECE573, Fall 2005 69

Parsing

Terminology

LL(1) Parsers

Overview of LR Parsing

70

Parsers: Terminology

- G : Grammar
- L(G): Language defined by G
- Vocabulary V of terminal (V_t) and non-terminal (V_n) symbols
- Strings are composed of symbols
- Productions (rewriting rules) tell how to derive strings (from other strings). We will use the **standard** BNF form.

ECE573, Fall 2005 71

Micro in Standard BNF

1	Program	::=	**BEGIN** Statement-list **END**
2	Statement-list	::=	Statement StatementTail
3	StatementTail	::=	Statement StatementTail
4	StatementTail	::=	λ
5	Statement	::=	**ID :=** Expression **;**
6	Statement	::=	**READ (** Id-list **) ;**
7	Statement	::=	**WRITE (** Expr-list **) ;**
8	Id-list	::=	**ID** IdTail
9	IdTail	::=	**, ID** IdTail
10	IdTail	::=	λ
11	Expr-list	::=	Expression ExprTail
12	ExprTail	::=	**,** Expression ExprTail
13	ExprTail	::=	λ
14	Expression	::=	Primary PrimaryTail
15	PrimaryTail	::=	Add-op Primary PrimaryTail
16	PrimaryTail	::=	λ
17	Primary	::=	**(** Expression **)**
18	Primary	::=	**ID**
19	Primary	::=	**INTLITERAL**
20	Add-op	::=	**PLUSOP**
21	Add-op	::=	**MINUSOP**
22	System-goal	::=	Program **SCANEOF**

Compare this to slide 30

A ::= B | C
↕
A ::= B
A ::= C

A ::= B {C}
↕
A ::= B tail
tail ::= C tail
tail ::= λ

::= and → are equivalent

ECE573, Fall 2005 72

Leftmost Derivation

- Rewriting of a given string starts with the leftmost symbol

Exercise: do a leftmost derivation of input program *F(V+V)* given the Grammar:

1: E → Prefix (E)
2: E → V Tail
3: Prefix → F
4: Prefix → λ
5: Tail → + E
6: Tail → λ

Draw the parse tree

ECE573, Fall 2005 73

Top-down and Bottom-up Parsers

- Top-down parsers use left-most derivation
- Bottom-up parsers use right-most derivation

Notation:

– LL(1) : Leftmost deriv. with 1 symbol lookahead
– LL(k) : Leftmost deriv. with k symbols lookahead
– LR(1) : Rightmost deriv. with 1 symbol lookahead

ECE573, Fall 2005 74

Grammar Analysis Algorithms

Follow (A) = $\{a \varepsilon V_t | S \Rightarrow^+ \ldots Aa \ldots\}$ or $\{\lambda$, if $S \Rightarrow^+ \ldots A\}$

In English: the follow set

- is the set of possible terminal symbols that can follow a given nonterminal.
- consists of all terminals that can come after A in *any* program that can be generated with the given grammar. It also includes λ, if A can be at the very end of any program.

First(α) = $\{a \varepsilon V_t | \alpha \Rightarrow^* a\beta\}$ or $\{\lambda$, if $\alpha \Rightarrow^* \lambda\}$

In English: the first set

- is the set of possible terminal symbols that can be at the beginning of the nonterminal A. It also includes λ, if A may produce the empty string.

S: start symbol of the grammar
a: a teminal symbol
A: a non-terminal symbol
α: any string

$\Rightarrow$ derived in 1 step
$\Rightarrow^+$ derived in 1 or more steps
$\Rightarrow^*$ derived in 0 or more steps

ECE573, Fall 2005 75

Towards Parser Generators

The main issue: as we read the source program tokens, we need to *decide what productions to use*.

Step 1: find the (lookahead) tokens that can tell that a production P (which has the form $A \rightarrow X_1 \ldots X_m$) applies

Predict(P) :
if not (λ in First($X_1 \ldots X_m$)) return First($X_1 \ldots X_m$)
else return (First($X_1 \ldots X_m$) - λ) $\cup$ Follow(A)

ECE573, Fall 2005 76

Parse Table

Step 2: building the parse table.

the parse table shows which production for a non-terminal V_n to take, given a terminal V_t

More formally:

$T : V_n \times V_t \longrightarrow P \cup \{Error\}$

ECE573, Fall 2005 77

Building the Parse Table

T[A][t] initialize all fields to "error"

Foreach A:

Foreach P with A on its LHS:

Foreach t in Predict(P) :

T[A][t] = P

Exercise: build the parse table for Micro

ECE573, Fall 2005 78

Building Recursive-Descent Parsers from LL(1) Parse Tables

Given the parse table we can create a program that writes the recursive descent parse procedures discussed earlier.

Remember the algorithm on page 34.

(If the choice of production is not unique, the parse table tells us which one to take.)

However there is an easier method...

A Stack-Based Parser Driver for LL(1)

Given the parse table, a stack-based algorithms looks much simpler than the generator of a recursive-descent parser.

The basic algorithm is

1 push the RHS of the production onto the stack
2 pop a symbol. If it's a terminal, match it;
3 if it's a non-terminal, take its production according to the parse table and goto 1

Algorithm on page 121

Including Semantic Actions in a Stack-Based Parser Generator

- Action symbols are simply pushed onto the stack as well.
- When popped, the semantic action routines are called.

ECE573, Fall 2005 81

Turning Non-LL(1) into LL(1) Grammar

consider :

stmt ::= if <expr> then <stmt list> endif

stmt ::= if <expr> then <stmt list> else <stmt list> end if

It is not LL(1) because it has a common prefix

We can turn this into:

stmt ::= if <expr> then <stmt list> <if suffix>

<if suffx> ::= end if

<if suffix> ::= else <stmt list> endif

ECE573, Fall 2005 82

Left-Recursion

E ::= E + T is left-recursive (the LHS is also the first symbol of the RHS)

How would the stack-based parser algorithm handle this production?

ECE573, Fall 2005 83

Removing Left Recursion

Example:

E → E + X
E → X

→

E → E1 Etail
E1 → X
Etail → + X Etail
Etail → λ

→ This can be simplified

(Algoritm on page 125)

ECE573, Fall 2005 84

If-Then-Else Problem

(a motivating example for LR grammars)

If *x* then *y* else *z*

If *a* then if *b* then *c* else *d*

this is analogous to a bracket notation when left brackets >= right brackets: [[ ]

Grammar:

$S \rightarrow [\ S\ C$

$S \rightarrow \lambda$

$C \rightarrow\ ]$

$C \rightarrow \lambda$

$[\ [\ ]\ \xrightarrow[\text{ambiguous}]{} SS\lambda C$ or $SSC\lambda$

ECE573, Fall 2005 85

Solving the If-Then-Else Problem

- The ambiguity exists at the language level as well. The semantics needs to be defined properly:

  e.g., "the *then* part belongs to the closest matching *if*"

$S \rightarrow [\ S$

$S \rightarrow S1$

$S1 \rightarrow [\ S1\ ]$

$S1 \rightarrow \lambda$

$\longrightarrow$ This grammer is still not LL(1), nor is it LL(k)
Show that this is so.

ECE573, Fall 2005 86

Parsing the If-Then-Else Construct

LL(k) parsers can look ahead k tokens.

(LL(k) will not be discussed.

Most important: be able to

- explain in English what LL(k) means and
- recognize a simple LL(k) grammar.)

For the If-Then-Else construct, a parsing strategy is needed that can **look ahead at the entire RHS** (not just k tokens) before deciding what production to take.

→ LR parsers can do that.

ECE573, Fall 2005 87

LR Parsers

A Shift-Reduce Parser:

- Basic idea: put tokens on a stack until an entire production is found.
- Issues:
  - recognize the end point of a production
  - find the length of the production (RHS)
  - find the corresponding nonterminal (i.e., the LHS of the production)

ECE573, Fall 2005 88

Data Structures for Shift-Reduce Parsers

At each state, given the next token,

- a *goto table* defines the successor state
- an *action table* defines whether to
  - shift (put the next state and token on the stack)
  - reduce (a RHS is found, process the production)
  - terminate (parsing is complete)

ECE573, Fall 2005 89

Example of Shift-Reduce Parsing

Consider the simple Grammar:

1: <program> → **begin** <stmts> **end** $

2: <stmts> → SimpleStmt ; <stmts>

3: <stmts> → **begin** <stmts> **end** ; <stmts>

4: <stmts> → λ

Shift Reduce Driver Algorithm on page 142, Fig 6.1..6.4

ECE573, Fall 2005 90

LR Parser Generators

(OR: HOW TO COME UP WITH GOTO AND ACTION TABLES?)

Basic idea:

- **Shift** in tokens; at any step keep the set of productions that match the tokens already read.
- **Reduce** RHS of recognized productions (i.e., replace them by their LHS)

LR(k) Parsers

LR(0) parsers:

- no lookahead
- *predict* which production to use by looking only at the symbols already read.

LR(k) parsers:

- k symbol lookahead
- most powerful class of deterministic bottom-up parsers

Terminology for LR Parsing

- Configuration:

  $A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j$

- Configuration set:

  all the configurations that apply at a given point in the parse. For example:

  $A \rightarrow B \bullet CD$
  $A \rightarrow B \bullet GH$
  $T \rightarrow B \bullet Z$

• marks the point to which the production has been recognized

ECE573, Fall 2005 93

Configuration Closure Set

- Include all configurations necessary to recognize the next symbol after the mark •
- For example:

$S \rightarrow E\$$
$E \rightarrow E + T \mid T$
$T \rightarrow ID \mid (E)$

closure0({$S \rightarrow \bullet E\$$})={
$S \rightarrow \bullet E\ \$$
$E \rightarrow \bullet E+T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet ID$
$T \rightarrow \bullet (E)$ }

ECE573, Fall 2005 94

Successor Configuration Set

- Starting with the initial configuration set
  s0 = closure0({S→ • α $}),

  a LR(0) parser will find the successor, given a (next) symbol X.

  X can be either a terminal (a token from the scanner) or a nonterminal (the result of a reduction)
- Determining the successor **s' = go_to(s,X) :**
  1. pick all configurations in s of the form A → β • X γ
  2. take closure0 of this set

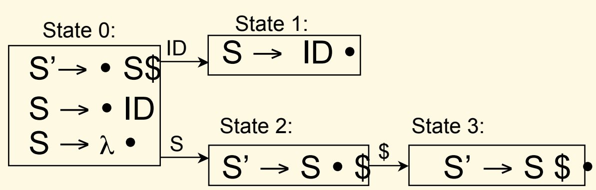
ECE573, Fall 2005 95

Building the Characteristic Finite State Machine (CFSM)

- Nodes are configuration sets
- Arcs are go_to relationships

Example:

1: S'→ S$
2: S→ ID
3: S→ λ



ECE573, Fall 2005 96

Building the go_to Table

- Building the go_to table is straightforward from the CFSM:

For the previous example the table looks like this:

State	Symbol: ID	Symbol: $	Symbol: S
0	1		2
1			
2		3	
3			

strictly speaking, State 0 is *inadequate,* i.e., there is a shift-reduce conflict. To resolve this conflict, An LR(1) parser is needed.

ECE573, Fall 2005 97

Building the Action Table

Given the configuration set s:

- We shift if the next token matches the terminal after the •

  in $A \rightarrow \alpha \bullet a \beta \in s$ and $a \in V_t$, else error

- We reduce i if the • is at the end of a production

  $B \rightarrow \alpha \bullet \in s$ and production i is $B \rightarrow \alpha$

ECE573, Fall 2005 98

LR(0) and LR(k) Grammars

- For LR(0) grammars the action table entries just described are unique.
- For most useful grammars we cannot decide on shift or reduce based on the symbols read. Instead, we have to look ahead *k* tokens. This leads to LR(k).
- However, it is possible to create an LR(0) grammar that is equivalent to any given LR(k) grammar (provided there is an end marker). This is only of theoretical interest because this grammar may be very complex and unreadable.

ECE573, Fall 2005 99

Exercise

- Create CFSM, go_to table, and action table for

S→ E$
E→ E + T | T
T→ ID | (E)

1: S→ E$
2: E→ E + T
3: E→ T
4: T→ ID
5: T→ (E)

ECE573, Fall 2005 100

LR(1) Parsing

- LR(0) parsers may generate
  - shift-reduce conflicts (both actions possible in same configuration set)
  - reduce-reduce conflicts (two or more reduce actions possible in same configuration set)
- The configurations for LR(1) are extended to include a lookahead symbol

$\boxed{A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j\ ,\ l}$ $\quad l \in V_t \cup \{\lambda\}$

Lookahead symbol

Configurations that differ only in the lookahead symbol are combined:

$\boxed{A \rightarrow X_1 \ldots X_i \bullet X_{i+1} \ldots X_j\ ,\ \{l_1 \ldots l_m\}}$

ECE573, Fall 2005 101

Configuration Set Closure for LR(1)

$S \rightarrow E\$$
$E \rightarrow E + T \mid T$
$T \rightarrow ID \mid (E)$

closure1($\{S \rightarrow \bullet E\$, \{\lambda\}\}$)={
$S \rightarrow \bullet E\$, \{\lambda\}$
$E \rightarrow \bullet E+T, \{\$+\}$
$E \rightarrow \bullet T, \{\$+\}$
$T \rightarrow \bullet ID, \{\$+\}$
$T \rightarrow \bullet (E), \{\$+\}$
}

ECE573, Fall 2005 102

Goto and Action Table for LR(1)

- The function goto1(configuration-set,symbol) is the same as goto0() for LR(0)
- Goto table is also created the same way as for LR(0)
  - The lookahead symbols are simply copied with the configurations, when creating the successor states.

  Notice that the lookahead symbols are a *subset* of the follow set.
- The Action table makes the difference. The lookahead symbol is used to decide if a reduction is applicable. Hence, the lookahead symbol resolves possible shift-reduce conflicts.

ECE573, Fall 2005 103

Example: LR(1) for G3

S→ E$
E→ E + T | T
T → T * P | P
P→ ID | (E)

- Exercise:
  - create states and the goto table
  - create the action table
  - explain how you see that this is LR(1) and not LR(0)

ECE573, Fall 2005 104

Problems with LR(1) Parsers

LR(1) parsers are very powerful. However,

- The table size can grow by a factor of $|V_t|$
- Storage-efficient representations are an important issue.

Example: Algol 60 (a simple language) includes several thousand states.

ECE573, Fall 2005 105

Solutions to the LR(1) Size Problem

Several parser schemes similar to LR(1) have been proposed

- LALR: merge certain states. There are several LR optimization techniques (will not be discussed further).
- SLR (simple LR): build a CFSM for LR(0) then add lookahead. Lookahead symbols are taken from the Follow sets of a production.

ECE573, Fall 2005 106

Exercise

- Determine if G3 is an SLR Grammar:

  Hint: the states 7 and 11 have shift-reduce conflicts. Can they be resolved by looking at the Follow set?

  (Remember the lookahead symbol sets is a subset of the follow set)

ECE573, Fall 2005 107

We have covered ...

- Scanners, scanner generators
- Parsers:
  - Parser terminology
  - LL(1) parsing and parser generation: building stack-based parsers, including action symbols.
  - Overview of LR parsers: shift-reduce parsers. CFSM. Basics of LR(1).

ECE573, Fall 2005 108

Semantic Processing

109

Some “Philosophy” About the Structure of Compilers at First.

ECE573, Fall 2005 110

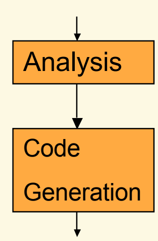
Properties of 1-Pass Compilers

- efficient
- coordination and communication of passes not an issue
- single traversal of source program restricts semantics checks and actions.
- no (or little) code optimization (peephole optimization can be added as a separate pass)
- difficult to retarget, architecture-dependent. Architecture-dependent and independent decisions are mixed.

ECE573, Fall 2005 111

1-Pass Analysis + 1-Code Generation Pass

- More machine independent
- Can add optimization pass
- There is an intermediate representation (IR, see slide 10) that represents the analyzed program. It is input to the code generator.
- Each pass can now be exchanged independently of each other



ECE573, Fall 2005 112

Multi-Pass Analysis

- Scanner can be a separate pass, writing a stream (file) of tokens.
- Parser can be a separate pass writing a stream of semantic actions.
- Analysis is very important in all optimizing compilers and in programming tools
- Advantages of Multi-Pass Analysis:
  - can handle Languages w/o variable declarations (need multi-pass analysis for static semantics checking)
  - no “forward declarations” necessary

Multi-Pass Synthesis

We view a compiler as performing two major tasks.

Analysis
understanding syntax and semantics of the source program.

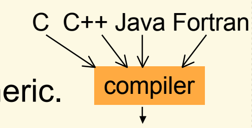
↓

Synthesis
generating the output (usually the target code)

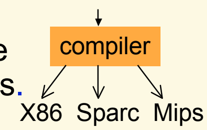
- Simple multi-pass synthesis: code-generation + peephole optimization
- Several optimization passes can be added
- Split into machine independent and dependent code generation phases is desirable
- Importance of early multi-pass compilers : space savings.

Families of Compilers

- Compilers that can understand multiple languages.
  - Syntax analysis has to be different.
  - Some program analysis passes are generic.
  - The choice of IR influences the range of analyzable languages.



- Compilers that generate code for multiple architectures.
  - Analysis and architecture-independent code generation can be the same for all machines.
  - Example: GNU C compiler. GCC uses two IRs: a tree-oriented IR and RTL.



Now the Specifics of Semantic Action Routines

A Common Compiler Structure: Semantic Actions Generate ASTs

- In many compilers, the sequence of semantic actions generated by the parser build an abstract syntax tree (*AST*, or simply *syntax tree*.)
- After this step, many compiler passes operate on the syntax tree.

ECE573, Fall 2005 117

Tree Traversals

After the AST has been built, it is traversed several times, for

- testing attributes of the tree (e.g., type checking)
- testing structural information (e.g., number of subroutine parameters)
- optimizations
- output generation.

ECE573, Fall 2005 118

Semantic Actions and LL/LR Parsers

- Actions are called either by parsing routines or by the parser driver. Both need provisions for semantic record parameter passing

Example:

passing semantic record

<if-stmt> → IF <expr> #start-if THEN <stmt-list> ENDIF #finish-if

- For LL parsers, semantic actions are perfect fits, thanks to their predictive nature
- In LR parsers, productions are only recognized at their end. It may be necessary to split a production, generating "semantics hooks"

<if-stmt> → <begin-if> THEN <stmt-list> ENDIF #finish-if
<begin-if> → IF <expr> #start-if

ECE573, Fall 2005 119

Semantic Records

or: how to simplify the management of semantic information

Idea: Every symbol (of a given production) has an associated storage item for semantic information, called semantic record.

- Semantic records may be empty (e.g., for ";" or <stmt-list>).
- Control statements often have 2 or more actions.
- Typically, semantic record information is generated by actions at symbols and is passed to actions at the end of productions.

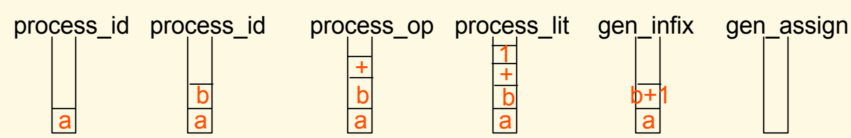
A good organization of the semantic records is the semantic stack.

ECE573, Fall 2005 120

Semantic Stack Example

- consider a:=b+1 *(Grammar on slide 40)*
- sequence of parse actions invoked:

  process_id, process_id, process_op, process_lit, gen_infix, gen_assign



ECE573, Fall 2005 121

Action-Controlled Semantic Stack

- Action routines can push/pop semantic records directly onto/from the stack.

This is called *action-controlled* stack.

- – Disadvantage: stack management has to be implemented in action routines by *you*, the compiler writer.

ECE573, Fall 2005 122

LR Parser-Controlled Stack

The idea:

- Every shift operation pushes a semantic record onto the semantic stack, describing the token.
- At a reduce operation, the production produces a semantic record and replaces all RHS records on the stack with it.

The effect of this:

- The action procedures don't see the stack. They only see the semantic records in the form of procedure parameters.
- Therefore, the user of a parser generator does not have to deal with semantic stack management. You only need to know that this is how the underlying implementation works.

Example: YACC

ECE573, Fall 2005 123

LL Parser-Controlled Stack

Remember: the parse stack contains predicted symbols, not the symbols already parsed.

- Entries for all RHS symbols (left-to-right) are also pushed onto the semantic stack and gradually filled in.
- When a production is matched: the RHS symbols are popped, the LHS symbol remains.
- Keep pointers to left,right,current,top symbol for each production in progress. Recursively store these values in a EOP (end of production) symbol as nonterminals on the RHS are parsed.
  - Algorithm and example on pages 238-241.

ECE573, Fall 2005 124

Symbol Tables

Operations on Symbol Tables:

- create table
- delete table
- enterId(tab,string) returns: entryId, exists
- find(tab,string) returns: entryId, exists
- deleteEntry(entryId)
- addAttributes(entryId,attributes)
- getAttributes(entryId) returns: attributes

Implementation Aspects of Symbol Tables

- Dynamic size is important. Space need can be from a few to tens of thousands of entries.

  Both should be provided:
  – dynamic growth for large programs
  – speed for small programs

Implementation Schemes

- Linear list
  - can be ordered or unordered
  - works for toy programs only
- Binary search trees
  - usually good solution. However, trees can be unbalanced, especially if alphabetical keys are used
- Hash tables
  - best variant. More complex. Good schemes exist
  - dynamic extension unclear
  - issues: clustering and deletion

Languages such as Java and C++ provide libraries!

ECE573, Fall 2005 127

Dealing with Long Identifiers

- can be a waste of space
- one solution is to store strings in a separate string array

i1.exp.the_weather_forecast_of_tomorrow.i.the_weather_forecast_of_today. ......

name length = 2	name length = 3	name length =32	name length = 1	name length =29	...
other attributes	other attributes	other attributes	other attributes	other attributes	

ECE573, Fall 2005 128

Symbol Table Issues

- Symbol tables can be one per program block
  - size can be smaller
  - issue of dynamic size still remains
  - deletion in hash tables is less of a problem
- Overloading (same name used for different identifiers)
  - keep symbols together. Context will choose between them
  - name “mangling” in C++

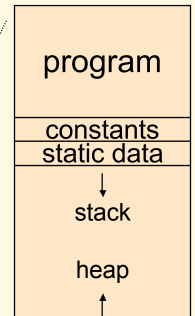
Symbol Table Attributes

- Examples:
  - Identifier and TypeDescriptor in Pascal (textbook p. 321/322)

Runtime Storage Organization

(remember this from your OS course?)

- Activation records (will be discussed later)
- Heap allocation
  - explicit malloc, free
  - implicit heap allocation (e.g., Lisp)
- Program layout in memory



- Procedure parameters (function pointers, formal procedures)

ECE573, Fall 2005 131

Processing Declarations

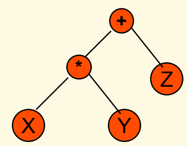
(overview)

- Attributes and implementation techniques of symbol tables and type descriptors
- Action routines for simple declarations
  - semantic routines for processing declarations and creating symbol table entries
- Action Routines for advanced features
  - constant declarations
  - enumeration types
  - subtypes
  - array types
  - variant records
  - pointers
  - packages and modules

ECE573, Fall 2005 132

Processing Expression and Data Structure References

- Simple identifiers and literal constants
- Expressions
  - Tree representations X*Y + Z



- Record/struct and array references

  A[i,j] → A + i**dim*_1 + j (if row major)

  R.f → R + *offset*(f)
- Strings
- Advanced features

ECE573, Fall 2005 133

Translating Control Structures

ECE573, Fall 2005 134

IF Statement Processing

IF-statement → **IF** #start B-expr #test **THEN** Stmts
{ **ELSIF** #jump #else_label B-expr #test
THEN Stmts }
Else-part
ENDIF #out_label

Else-part → **ELSE** #jump #else_label Stmts

Else-part → #else_label

Semantic record :

```
struct if_stmt {
   string  out_label;
   string  next_else_label;
}
```

ECE573, Fall 2005 135

Code for IF statement

```
          Evaluate B-expr1
          beq  res1 Else1
          Code for  Stmts1
          jmp Endif
Else1:
          Evaluate B-expr2
          beq  res2 Else2
          Code for  Stmts2
          jmp Endif
Else2:
          . . .
ElseN-1:
          Evaluate B-exprN
          beq  resN ElseN
          Code for  StmtsN
          jmp Endif
ElseN:
          Code for StmtsN+1
Endif:
```

Only blue code is generated by IF construct action routines

ECE573, Fall 2005 136

Loop Processing

While-Stmt → **WHILE** #start B-expr #test
LOOP Stmts **ENDLOOP** #finish

Semantic record :

```
struct while_stmt {
   string  top_label;
   string  out_label;
}
```

For-Stmt → **FOR** Id #enter **IN** Range
#init **LOOP** Stmts **ENDLOOP** #finish

Semantic record :

```
struct for_stmt {
   data_object id;
   data_object limit_val;
   string next_label, out_label;
(  boolean reverse_flag;  )
}
```

ECE573, Fall 2005 137

Code for WHILE statement

```
BeginWhile:
            Evaluate B-expr
            beq  res1 EndWhile
            Code for  Stmts
            jmp BeginWhile
EndWhile:
```

Only blue code is generated by IF construct action routines

ECE573, Fall 2005 138

Code for count-up FOR statement

```
        compute LowerBound
        compute UpperBound
        cmp LowerBound UpperBound res1
        bgt res1 EndFor
        index = LowerBound
        limit = UpperBound
Loop:
        Code for  Stmts
        cmp index limit res2
        beq res2 EndFor
        inc index
        jmp Loop
EndFor:
```

Only blue code is generated by IF construct action routines

ECE573, Fall 2005 139

CASE Statement Processing

Case-Stmt → **CASE** Expr #start **IS** When-list Others-option **ENDCASE ;** #finish_case

When-list → { **WHEN** Choice-list **:** Stmts #finish-choice }

Others-option → **ELSE** #start_others **:** Stmts #finish-choice

Others-option → #no_others

Choice-list → Choice { | Choice }

Choice → Expr #append_val

Choice → Expr **..** Expr #append_range

ECE573, Fall 2005 140

Code for CASE statement

```
             Evaluate Expr
             cmp Expr MinChoice res1                  #start
             blt  res1 Others
             cmp Expr MaxChoice res2
             bgt  res2 Others
             jumpx Expr Table-MinChoice               #append_val
L1:          Code for  Stmts1
             jmp EndCase                              #finish_choice
             . . .
LN:          Code for  StmtsN                         #append_val
             jmp EndCase                              #finish_choice
Others:      Code for  Stmts in Else clause           #start_others
             jmp EndCase                              #finish_choice
Table:       jmp L1
             . . .  (jmp Lx or jmp Others)            #finish_case
             jmp LN
EndCase:
```

Only blue code is generated by IF construct action routines

Finish_case needs to back patch here

ECE573, Fall 2005 141

Semantic Record for CASE statement

```
struct case_rec {
    struct type_ref  index_type;
    list_of_choice  choice_list;
  /* address of the JUMPX tuple (for back patching):    */
    tuple_index  jump_tuple;
  /* target of branches out:*/
    string  out_label;
  /* label of the code for ELSE clause: */
    string others_label;
}
```

ECE573, Fall 2005 142

Code Generation for Subroutine Calls

Parameter Types
Activation Records
Parameter Passing
Code Examples

Parameter Types

- Value Parameters :
  - copy at subroutine call. For large objects this can be done by either the caller or the callee.
  - an expression can be passed
- Result Parameters:
  - are copied at the end of the subroutine to return values to the caller
- Value-Result Parameters:
  - “copy-in-copy-out”. Enhances locality.

Parameter Types (2)

- Reference (var) parameters:
  - the address is passed in to the subroutine.
  - this is different from value-result, although for the user the semantics may look the same.
- Read-Only parameters:
  - small objects are passed by value, large parameters are passed by reference.

Dope Vectors

Additional information - no seen by the programmer - about parameters may need to be passed into subroutines, for example:

- bounds (on the parameter value)
- length (of a string or vector)
- storage allocation information
- data allocation information

Good compile-time analysis can reduce the need for passing dope vector information

Saving Registers

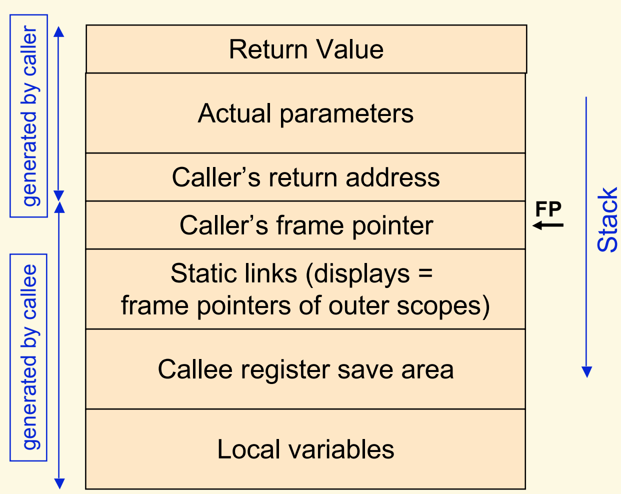
- Subroutines generally don't know which registers are in use by the caller. Solutions:
  - caller saves all used registers before call
  - callee saves the registers it uses
  - caller passes to the callee a bit vector describing used registers *(good only if hardware supported).*

Simple optimizations are useful (e.g., don't save registers if called subroutine does not use any registers)

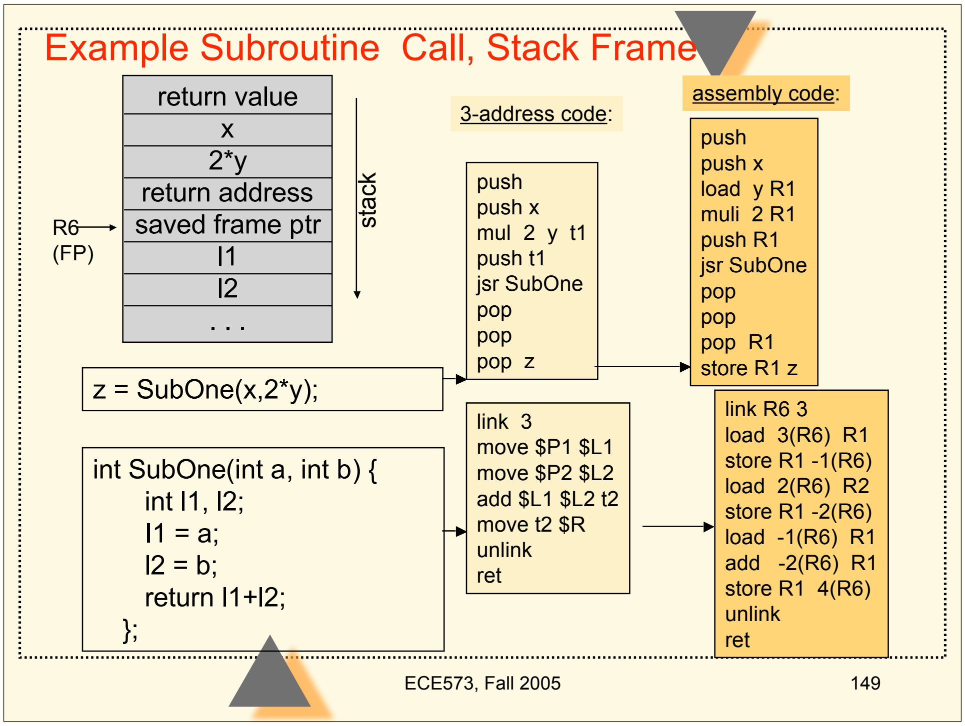
ECE573, Fall 2005 147

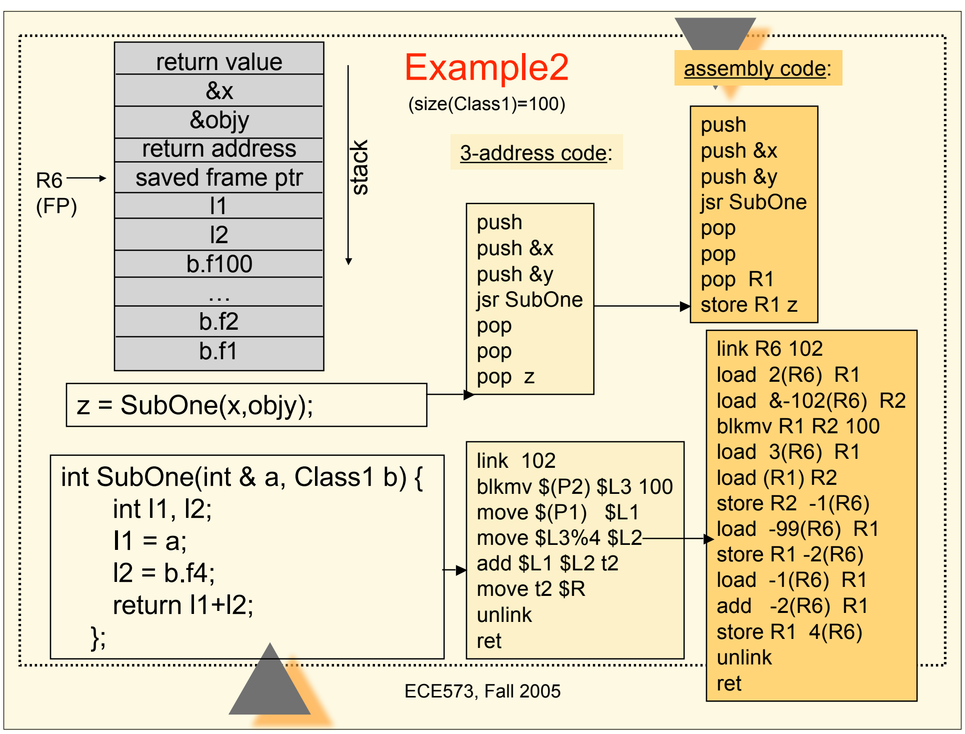
Activation Records

A typical activation record (or stack frame)



ECE573, Fall 2005 148


Example Subroutine Call, Stack Frame
return value
x
2*y
return address
saved frame ptr
l1
l2
. . .
R6
(FP)
stack
3-address code:
push
push x
mul 2 y t1
push t1
jsr SubOne
pop
pop
pop z
assembly code:
push
push x
load y R1
muli 2 R1
push R1
jsr SubOne
pop
pop
pop R1
store R1 z
z = SubOne(x,2*y);
int SubOne(int a, int b) {
int l1, l2;
l1 = a;
l2 = b;
return l1+l2;
};
link 3
move $P1 $L1
move $P2 $L2
add $L1 $L2 t2
move t2 $R
unlink
ret
link R6 3
load 3(R6) R1
store R1 -1(R6)
load 2(R6) R2
store R1 -2(R6)
load -1(R6) R1
add -2(R6) R1
store R1 4(R6)
unlink
ret
ECE573, Fall 2005
149


Example2
(size(Class1)=100)
return value
&x
&objy
return address
saved frame ptr
l1
l2
b.f100
…
b.f2
b.f1
R6
(FP)
stack
3-address code:
push
push &x
push &y
jsr SubOne
pop
pop
pop z
assembly code:
push
push &x
push &y
jsr SubOne
pop
pop
pop R1
store R1 z
z = SubOne(x,objy);
int SubOne(int & a, Class1 b) {
int l1, l2;
l1 = a;
l2 = b.f4;
return l1+l2;
};
link 102
blkmv $(P2) $L3 100
move $(P1) $L1
move $L3%4 $L2
add $L1 $L2 t2
move t2 $R
unlink
ret
link R6 102
load 2(R6) R1
load &-102(R6) R2
blkmv R1 R2 100
load 3(R6) R1
load (R1) R2
store R2 -1(R6)
load -99(R6) R1
store R1 -2(R6)
load -1(R6) R1
add -2(R6) R1
store R1 4(R6)
unlink
ret
ECE573, Fall 2005

Static Allocation of Activation Records

- Dynamic setup of activation records takes significant time (for short subroutines).
- Instead of on the stack, the compiler can allocate local variables and subroutine parameters in static memory locations.
- This will not work for recursive and parallel code (reentrancy is important in both cases)

ECE573, Fall 2005 151

Code Generation and Optimization

152

Local versus Global Optimization

- Local optimizations:

  operation is within basic block (BB).

  – A BB is a section of code without branches (except possibly at the end)

  – BBs can be from a few instructions to several hundred instructions long.

- Global optimizations will be introduced later.

Assembly Code Generation

- A simple code generation approach:

  macro-expansion of IR tuples

  Each tuple produces code independently of its context:

  advantage: simple, straightforward, easy to debug

  disadvantage: no optimization

  E.g., (+,a,b,c) generates store C

  (+c,d,e) generates a (redundant) load C

  Peephole optimizations help a little

Peephole Optimizations

- Simple pattern-match optimizations usually following a simple code generator.
  e.g., pattern: *store R X*, followed by *load R X*
  → delete *load R X*
- Can recognize patterns that can be performed by special instructions (machine-specific).
  e.g., pattern: *sub 1 R, jgt label*
  → replace by *sbr R label*

ECE573, Fall 2005 155

Peephole Optimizations

- Constant folding:
  - ADD lit1 lit2 result ⇒ MOVE <u>lit1+lit2</u> result
  - MOVE lit1 res1 ⇒ MOVE lit1 res1
    ADD lit2 res1 res2 MOVE <u>lit1+lit2</u> res2
- Strength reduction
  - MUL op 2 res ⇒ SHIFTL op 1 res
  - MUL op 4 res ⇒ SHIFTL op 2 res
- Null sequences
  - ADD op 0 res ⇒ MOVE op res
  - MUL op 1 res ⇒ MOVE op res
- Combine operations
  - MOVE A R_i ; MOVE A+1 R_{i+1} ⇒ DBLMOVE A R_i
  - JEQ L1 ; JMP L2 ; L1: ⇒ JNE L2

ECE573, Fall 2005 156

More Peephole Optimizations

- Simplify by algebraic laws
  - ADD lit op res ⇒ ADD op lit res
  - SUB op 0 res ⇒ NEG op res
- Special case instructions
  - SUB 1 R ⇒ DEC R
  - ADD 1 R ⇒ INC R
  - MOVE 0 R ; MOVE R A ⇒ CLR A
- Address mode operations
  - MOVE A R1 ; ADD 0(R1) R2 ⇒ ADD @A R2
  - SUB 2 R1 ; CLR 0(R1) ⇒ CLR --(R1)

ECE573, Fall 2005 157

Better Code Generation Schemes

- Keep “state” information

  IR tuples → $\boxed{\text{State machine}}$ → code

  an input IR tuple just changes the state. Code is generated as necessary when the machine changes state
- Generate code for an IR subtree at once
- Template matching, code generation for entire template

ECE573, Fall 2005 158

Code Generation steps

- 4 steps:
  - – instruction selection
    - this is very machine-specific. Some machines may provide complex instructions that perform 2 or more tuple (3-address) operations.
  - – address mode selection
  - – register allocation
  - – code scheduling *(not in text book)*

We will focus on these two topics

in reality, these tasks are intertwined

ECE573, Fall 2005 159

Address Mode Selection

- Even a simple instruction may have a large set of possible address modes and combinations. For example:
- Add a b c

can be indexed, indirect, live register, unassigned register

can be literal, indexed, live register, dead register

There are more than 100 combinations

ECE573, Fall 2005 160

More Choices for Address Mode

- Auto increment, decrement
- Three-address instructions
- Distinct address and data registers
- Specialized registers
- "Free" addition in indexed mode:

    MOVE (Reg)offset

  (This is very useful for subscript operations)

ECE573, Fall 2005 161

The textbook discusses Common Subexpression Elimination and Aliasing at this point.

These topics will be discussed later.

ECE573, Fall 2005 162

Register Allocation Issues

- 1. Eliminate register loads and stores

store R3,A

…

load R4,A

we want to recognize that R3 could be reused

- 2. Reduce register spilling.
  - Ideally all data is kept in registers until the end of the basic block. However, there may not be enough registers.

  What registers should be freed? ←**THE key question**

Optimal solutions are NP-complete problems

ECE573, Fall 2005 163

Register Allocation Terminology

- Registers can be:
  - unallocated: carry no value
  - live: carry a value that will be used later
  - dead: carry a value that is no longer needed
- Register association lists:

  variables (including temporaries) that are associated with a register can be
  - live (L, used again in the basic block before changed) or dead(D)
  - to be saved(S) at the end of the BB or not to be saved (NS)
    - corresponds to "dirty" attribute in previous algorithm
- Liveness Analysis of Variables:
  - a backwards pass through the code, detecting use and definition points to determine these attributes.

ECE573, Fall 2005 164

When to free a register?

- Assume a cost function for register and memory references. E.g., memory ref: 2, register ref: 1
- Freeing costs:
  - 0 (D,NS), (D,S) (no disadvantage in saving right away)
  - 2 (L,NS) (will need to reload later)
  - 4 (L,S) (store now, reload later)
- When a register is needed, look for the cheapest. If same cost, free the one with the most distant use, then load the new value and set the status to (L,NS) or (D,NS)
  - Note: Assignment to a variable makes previous status (D,NS)
- This cost may also be used to choose between code generation alternatives, e.g., commutative operations.
- Algorithms on pages 564 .. 566

Register Allocation

An example without optimized register allocation

A := B*C + D*E
D := C+(D-B)
F := E+A+C
A := D+E

```
1. (*,B,C,T1)        8. (+,E,A,T6)
2. (*,D,E,T2)        9. (+,T6,C,T7)
3. (+,T1,T2,T3)      10.(:=,T7,F)
4. (:=,T3,A)
                     11.(+,D,E,T8)
5. (-,D,B,T4)        12.(:=,T8,A)
6. (+,C,T4,T5)
7. (:=,T5,D)
```

```
Load  B,R1      Load D,R1      Load  E,R1     Load  D,R1
*     C,R1      -    B,R1      +     A,R1     +     E,R1
Load  D,R2      Load C,R2      +     C,R1     Store A,R1
*     E,R2      +    R1,R2     Store F,R1
+     R2,R1     Store D,R2
Store A,R1
```

Register Allocation Exercise

Optimized register allocation,
textbook, p 568

reduces the cost of storage-to-register and register-to-register operations from 34 to 25

Aliasing: A Problem for Many Optimizations

- A big problem in compiler optimizations is to recognize *aliases*.
- Aliases are "different names for the same storage location"
- Aliases can occur in the following situations
  - pointers may refer to the same variable
  - arrays may reference the same element
  - subroutines may pass in the same variable under two different names
  - subroutines may have side effects
  - Explicit storage overlapping
- The ramification here is, that we cannot be sure that variables hold the values they appear to hold. We need to conservatively mark values as killed.

Aliasing and Register Allocation

- on load of a variable *x*:
  for each variable aliased to *x* that is on a register association list: save it. (so that we are guaranteed to load the correct value)
- on store of a variable *x*:
  for each variable aliased to *x* that is on a register association list: remove it from the list. (so that we will not use a stale value later on)
- Analysis:
  - Most conservative: all variables are aliased
  - Less conservative: name-only analysis
  - Advanced: array subscript analysis, pointer analysis

  At subroutine boundaries: often conservative analysis. All (global and parameter) variables are assumed to be aliased.

ECE573, Fall 2005 169

Virtual Register Allocation

A register allocation algorithm can start from two possible situations:

1. All variables are in memory (this is the case when starting from 3-address code) -- the textbook algorithm starts from this point
2. Variables are placed in virtual registers -- the Cooper/Torczon algorithms have this starting point

> Allocation of virtual registers is easy:
>
> Whenever a new register is needed, an additional register number is taken.
>
> Move memory to register: either before the first use or at the beginning of the BB
>
> Move register to memory: at the end of the BB if the register has been written to

Virtual Register allocation is also necessary when performing code scheduling before register allocation -- Explain why.

ECE573, Fall 2005 170

Top-Down Register Allocation

(A Simple Algorithm by Cooper/Torczon 625)

- Basic idea:

  In each basic block (BB) do this:
  - find the number of references to each variable
  - assign available registers to variables with the most references

  Details:
  - keep some free registers for operations on unassigned variables
  - store *dirty* registers at the end of the BB. Do this only for variables (not for temporaries)
    - not doing this for temporaries exploits the fact that they are never live-out of a block. This is knowledge that would otherwise need global analysis.

ECE573, Fall 2005 171

Bottom-Up Register Allocation

(A Better Algorithm by Cooper/Torczon p. 626)

for each tuple *op A B C* in a BB do :
    r_x = ensure(A)   // make sure A is in a register
    r_y = ensure(B)   // make sure B is in a register
    if r_x is no more used then free(r_x)
    if r_y is no more used then free(r_y)
    r_z = allocate(C)   // make a register available for C
    mark r_z dirty
    generate(op,r_x,r_y,r_z)   // emit the actual code
for each dirty register *r* do :
    generate(“move”,r,r→opr())

Cooper/Torczon’s algorithm assumes A,B,C are virtual registers. We will assume they are variables.

ECE573, Fall 2005 172

Bottom-Up Register Allocation continued

ensure(opr)
if opr is already in a register *r* then
  return *r*
else
  r = allocate(opr)
  generate("move",opr,*r*)
  return *r*

free(r)
if r is dirty then
  generate("move",r,r→opr())
mark r free

allocate(opr)
if there is a free register *r* then
  take *r*
else
  find *r* with the most distant next use
  free (r)
  mark r associated with opr;
return *r*

Next_use analysis:
one backward pass through the BB is sufficient.

ECE573, Fall 2005 173

Other Register Allocation Schemes

Variations of the presented scheme:

- consider more than one future use
- register "coloring"
- better cost model: consider instruction size and timing; factor in *storage-to-register* instructions
- include more address modes
- include *register-to-register* moves
- consider peephole optimizations

Register allocation is still a research area.

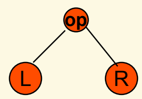
ECE573, Fall 2005 174

Context-sensitive Code Generation

(considering a larger window of code, but still within a basic block)

Generating code from IR trees.

Idea:



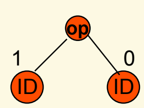
if evaluating R takes more registers than L, it is better to

- evaluate R
- save result in a register
- evaluate L
- do the (binary) operation

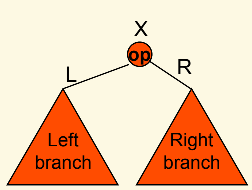
ECE573, Fall 2005 175

Determining Register Needs

Assuming register-to-register and storage-to register instructions



For ID nodes (these are leaf nodes):
- left: 1 register
- right: 0 registers



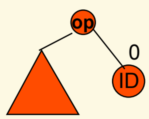
Register need of the combined tree:
X =
- L+1 , if R = L
- max(R,L) , if R ≠ L

ECE573, Fall 2005 176

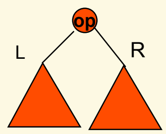
Algorithm for Code Generation Using Register-Need Annotations

Recursive tree algorithm. Each step leaves result in R1

(R1 is the first register in the list of available registers)



Case 1: right branch is an ID:
- generate code for left branch
- generate OP ID,R1 (op,R1,ID,R1)

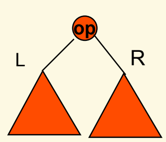


Case 2: min(L,R) >= max available registers:
- generate code for right branch
- spill R1 into a temporary T
- generate code for left branch
- generate OP T,R1

ECE573, Fall 2005 177

Tree Code Generation continued

Remaining cases: at least one branch needs fewer registers than available



min(R,L) < available regs

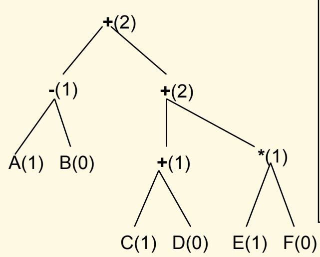
Case 3: R < max available registers:
- generate code for left branch
- remove first register (R1) from available register list
- generate code for right branch (result in R2)
- generate OP R2,R1

Case 4: L < max available registers:
- temporarily swap R1 and R2
- generate code for right branch
- remove first register (R2) from available register list
- generate code for left branch (result in R1)
- generate OP R2,R1

ECE573, Fall 2005 178

Example Tree Code Generation

(A-B)+((C+D)+(E*F))



	Ra holds	Rb holds	available regs.
Load C,Rb	--	C	Ra Rb
Add D,Rb	--	C+D	Rb Ra
Load E,Ra	E	C+D	Rb Ra
Mult F,Ra	E*F	C+D	Ra
Add Ra,Rb	--	C+D+E*F	Ra
Load A,Ra	A	C+D+E*F	Rb Ra
Sub B,Ra	A-B	C+D+E*F	Ra
Add Rb,Ra	A-B+C+D+E*F	--	Ra
			Ra Rb

ECE573, Fall 2005 179

Code Scheduling

- Motivation:

  processors can overlap the execution of consecutive instructions, but only if they are not dependent on each other

mult R2,R3		load X,R0
load X,R0	→	mult R2,R3
add R0,R4		add R0,R4

- Problem:

  this is not independent of the other register generation issues. For example: reordering instructions may create register conflicts

ECE573, Fall 2005 180

Processor Models for Code Scheduling

1. Processor enforces dependences.

   Compiler reorders instructions as much as possible

   ⇒Processor guarantees correctness

2. Processor assumes that all operands are available when instruction starts

   Compiler inserts NOPs to create necessary delays

   ⇒Compiler guarantees correctness

ECE573, Fall 2005 181

Code Scheduling Goal

- Annotate each operation with the cycle in which it can start to execute
  - operations can execute as soon as their operands are available
  - each operation has a delay, after which its result operand becomes available
  - the processor architecture defines how many and what type of operations can start in the same cycle
- Minimize the time until all operations complete

ECE573, Fall 2005 182

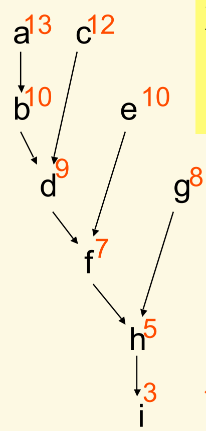
Precedence Graph

- shows operand dependencies of operations
- may also show anti-dependences on registers
  - anti-dependence: an operation that reuses a register must wait for the completion of the previous use of this register
  - anti-dependences may be removed by renaming registers
- can be annotated to show cumulative latencies

ECE573, Fall 2005 183

Precedence Graph Example

a: loadAI $r_0, 0 \Rightarrow r_1$
b: add $r_1, r_1 \Rightarrow r_1$
c: loadAI $r_0, 8 \Rightarrow r_2$
d: mult $r_1, r_2 \Rightarrow r_1$
e: loadAI $r_0, 16 \Rightarrow r_2$
f: mult $r_1, r_2 \Rightarrow r_1$
g: loadAI $r_0, 24 \Rightarrow r_2$
h: mult $r_1, r_2 \Rightarrow r_1$
i: storeAI $r_1 \Rightarrow r_0, 0$



<u>Weights (=latencies)</u>
memory op: 3
mult: 2
others: 1

← operation must start no later than 3 cycles before end of block

ECE573, Fall 2005 184

Precedence Graph Example: Removing Anti-Dependences

The graph on the previous slide does not show anti dependences. Here's how to remove them:

			→		
a: loadAI	$r_0, 0 \Rightarrow r_1$			a: loadAI	$r_0, 0 \Rightarrow r_1$
b: add	$r_1, r_1 \Rightarrow r_1$			b: add	$r_1, r_1 \Rightarrow r_2$
c: loadAI	$r_0, 8 \Rightarrow r_2$			c: loadAI	$r_0, 8 \Rightarrow r_3$
d: mult	$r_1, r_2 \Rightarrow r_1$			d: mult	$r_2, r_3 \Rightarrow r_4$
e: loadAI	$r_0, 16 \Rightarrow r_2$			e: loadAI	$r_0, 16 \Rightarrow r_5$
f: mult	$r_1, r_2 \Rightarrow r_1$			f: mult	$r_4, r_5 \Rightarrow r_6$
g: loadAI	$r_0, 24 \Rightarrow r_2$			g: loadAI	$r_0, 24 \Rightarrow r_7$
h: mult	$r_1, r_2 \Rightarrow r_1$			h: mult	$r_6, r_7 \Rightarrow r_8$
i: storeAI	$r_1 \Rightarrow r_0, 0$			i: storeAI	$r_8 \Rightarrow r_0, 0$

Note, register allocation and scheduling have conflicting demands. Ideally, the two techniques should be applied together. However, due to their complexity, most compilers separate them.

ECE573, Fall 2005 185

Local List Scheduling

- local = within a basic block
- outline of the algorithm:
  1. rename registers to remove anti-dependences
  2. build precedence graph
  3. assign priorities to operations
     We use the cumulative latency as the priority
  4. iteratively select an operation and schedule it

What makes scheduling difficult?

ECE573, Fall 2005 186

List Scheduling Algorithm

```
Cycle ← 1
Ready ← leaves of P
Active ← Ø
while (Ready∪Active≠ Ø)
  if Ready ≠ Ø then
    remove an op from Ready
    S(op) ← Cycle
    Active ← Active ∪ op
  Cycle ← Cycle + 1

  for each op ∈ Active
    if S(op)+delay(op) ≤ Cycle then
      remove op from Active
      for each successor s of op in P
        if s is ready then
          Ready ← Ready ∪  s
```

P: the precedence graph

Ready: list of operations ready to be scheduled

Active: operations being executed (scheduled, not yet completed)

delay(op): execution time of op

S(op): start time of op

ECE573, Fall 2005 187

Alternative List Scheduling Schemes

- Priority Schemes make a big difference. Possible Priorities:
  - longest path that contains an op
  - number of immediate successors
  - number of descendants
  - latency of operation
  - increase priority for last use of a value
- Forward versus backward scheduling

ECE573, Fall 2005 188

Coordination Schemes for Register Allocation and Instruction Scheduling

Scheme 1:

- Generate 3-address code
- Generate code, using any number of registers
- Instruction scheduling
  - List scheduling. Use precedence graph ***with*** removing anti-dependences
- Register allocation
  - using the unmodified Cooper/Torczon bottom-up *register allocation* algorithm.

Scheme 2:

- Generate 3-address code
- Register allocation
  - using the textbook *register tracking* or the modified *bottom-up Cooper/Torczon* algorithm.
- Instruction scheduling
  - List scheduling. Use precedence graph ***without*** removing anti-dependences

Global Program Optimization and Analysis

Motivation

- *Local* register allocation is not optimal
  - All dirty registers are saved at the end of the basic block
- What is missing is information about the flow of information across basic blocks
  - Values may already be in registers at the beginning of the block
  - Value may be reused in the next block
- Solution approaches:
  - Compute the LiveOut set of variables
  - Deal with the difficulties
    - There must be coordination of register use across blocks
    - Define what you mean by "next use" if it is in a different block

→ This leads to *global* register allocation, discussed later

ECE573, Fall 2005 191

Introductory Remarks

- What is an optimization
- Interdependence of optimizations
- What IR is best for optimizations?
- What improvements can we expect from optimizations? Does it always improve?
- What is an optimizing compiler?
- Analysis versus Transformation

ECE573, Fall 2005 192

What is an Optimization?

Criterion 1: Code change must be safe

An optimizations must not change the answer (the result) of the program. This can be subtle:

– Is it safe to do this move?

```
DO i=1,n
 <loop-invariant expression>
 ...
ENDDO
```

What if the expression is a/n ?

– Code size can be important. Optimizations that increase the code size may be considered unsafe (we will ignore this for now, however)

ECE573, Fall 2005 193

What is an Optimization?

Criterion 2: Code change must be profitable

- The performance of the transformed program must be better than before.

  This is sometimes difficult to determine, because:

  – the compiler does not have enough information about machine costs, or it knows only average costs.

  – the compiler does not have sufficient information about program input data.

  – the compiler may not have sufficiently powerful analysis techniques.

  Sometimes profiling is used to alleviate these problems. Profiling works only for some **average case!**

- The code size must be smaller (not always important)

ECE573, Fall 2005 194

Interdependence Of Optimizations

Usually, optimizations are applied one-by-one.

In reality they are interdependent.

For example:

```
a = 3
b = 0
IF (b == a-2)
  a = 5
ENDIF

IF (a == 3)
  print “success”
ELSE
  print “failure”
ENDIF
```

ECE573, Fall 2005 195

Source and Code-level Optimizations

- Examples of source-level optimizations:
  - eliminating unreachable code
  - constant propagation (is also an analysis technique)
  - loop unrolling (may also be done at instruction level)
  - eliminating redundant bound checks
  - loop tiling
  - subroutine inline expansion (may not be an optimization)
- Examples of code-level optimizations:
  - register allocation
  - thorough use of instruction set and address modes
  - cache and pipeline optimizations
  - instruction-level parallelization
  - strength reduction (may also be done at source level)

ECE573, Fall 2005 196

Compiler Optimizations in Perspective

- gain from (sequential program) optimizations :
  - 25% - 50%
- gain from parallelization:
  - 0-1000%
- gain from (manually) improved algorithms: 0 - ?

  e.g. replacing a $10*n^3$ by a $50*n^2$ algorithm
    - n=5 : no gain
    - n=100 : 500-fold improvement
- important: some optimization techniques may *decrease* performance in some code patterns!

ECE573, Fall 2005 197

Optimizing Compilers

- Term is used for compilers that use more than local, basic block optimizations. They include some form of global program analysis (analysis beyond basic blocks, sometimes beyond individual subroutines).
- Optimizations are time-consuming. Apply them where the return is biggest:
  - in loops (repetitive program sections)
  - at subroutine calls
  - in frequently executed code (look at profile)
    - "90/10 rule": 10 % of the loops contain 90% of the execution time

ECE573, Fall 2005 198

The Role of Program Analysis

Program analysis must precede many optimizations.

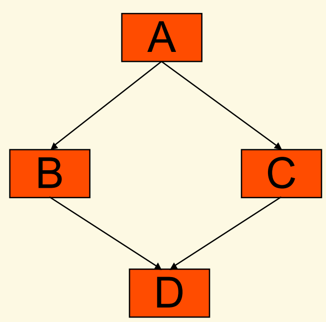
- Control flow analysis determines where program execution goes next
- Data flow analysis determines how program variables are affected by program sections
- Data-dependence analysis determines which data references in a program access the same storage location.

(Sometimes *data flow analysis* is used as a generic term for all these analyses)

ECE573, Fall 2005 199

Control Flow Analysis

```
A
IF (cond) THEN
  B
ELSE
  C
ENDIF
D
```



Control-Flow Graph
(Text-book calls it Data-Flow Graph)

ECE573, Fall 2005 200

Control Flow Analysis

Control flow imposes dependences

- the execution of a code block must wait until the control conditions have been evaluated.
- This is true even if there are no data dependences.

Exercise: draw the control flow graph of this program segment

```
d=100
IF (a==b) THEN c=d+e
ELSE
  IF (c==e) THEN d=0
ENDIF
b=b*2
DO i=d,e
  a[i]=0
ENDDO
```

ECE573, Fall 2005 201

Interprocedural Analysis

A small detour before we proceed with transformations and analysis methods

202

A small detour through

Interprocedural Analysis

Analysis across subroutines is an even bigger issue than analysis across basic blocks.

Approaches:

- Subroutine inline expansion (a.k.a. inlining)
  - Saves call overhead for small subroutines.
  - Eliminates the need for interprocedural analysis
- Interprocedural analysis - extend analysis to traverse all routines
  - Eliminates (or reduces) conservative assumptions at subroutine boundaries, such as
    - the assumption that all variables seen by a subroutine (parameters, global variables) are read and written.
    - the assumption that all subexpressions are killed

ECE573, Fall 2005 203

A small detour through

Interprocedural Analysis

IPA propagates knowledge gathered in one subroutine to the others.

This analysis may need to be iterative.

Example:
constant propagation

```
Main Program
a = 3
call sub1(a)
call sub2(a)
```

```
subroutine sub1(x)
  ...
  localvar = x
  ...
```

```
subroutine sub2(y)
  ...
  localvar = y
  ...
```

ECE573, Fall 2005 204

A small detour through Interprocedural Analysis:

Interprocedural Dataflow Analysis

Building Definition and Use sets

- Def set: the set of variables written to
- Use set: the set of variables read

these sets are used by many optimizations

Given LocalUse and LocalDef of each subroutine:

$$Use(P) = LocalUse(P) \bigcup_{Q \in called(P)} Use(Q)$$

$$Def(P) = LocalDef(P) \bigcup_{Q \in called(P)} Def(Q)$$

ECE573, Fall 2005 205

A small detour through Interprocedural Analysis:

Finding Local Def, Use

- Determining Def and Use sets within a subroutine is easy.

  Exercise: determine Def, Use sets for

```
c = d+e
d = a[j]
If (a<b) then
  a[j] = sb[i,k]
   *p = *q
endif
```

- In the presence of control flow one can analyze *may* or *must* definitions and uses.
  - Simple analysis: *may* def/use analysis
  - More advanced: *flow-sensitive* analysis

ECE573, Fall 2005 206

A small detour through Interprocedural Analysis:

Algorithm for Computing Def, Use Sets Interprocedurally

- In non-recursive programs:
  - compute Def, Use sets bottom-up in call tree
- In recursive programs:
  - iterate until the sets don't change any more
  - (Algorithm on page 632)

ECE573, Fall 2005 207

A small detour through Interprocedural Analysis:

Exercise

- find Def, Use sets for program on p. 632
- how does this information help the program analysis?

ECE573, Fall 2005 208

Three Basic Optimizations

- Common subexpression elimination
- Factoring loop-invariant expressions
- Strength reduction

209

Common Subexpression Elimination

An optimization that can simplify the generated code significantly

CSE removes redundant computation

1: A = B+C*D

2: E = B+C*D

keep the result in a temporary and reuse for stmt 2

Difficulty: recognize when the expression is "killed"

1: A = B+C*D
B = <new value>
2: E = B+C*D

B is **killed**. The expression it held is no longer "(a)live".

ECE573, Fall 2005 210

Value Numbering
(used by CSE)

Give unique numbers to **expressions** that are computed from the same operands X

Operands are the same if

1. Their name is the same
2. They were not modified since the previous expression was computed

The Value Numbering compiler algorithm keeps a "last defined" attribute for every variable and for every temporary holding an expression.

The Use of Value Numbering in CSE is obvious:

A temporary holding the result of expression_1 can be reused in place of expression_2 if the two expressions have the same value number.

ECE573, Fall 2005 211

When to Apply CSE in the Compiler?

- Option 1: when generating code
  - suppress code generation for such subexpressions and use the temporary holding the needed value instead, or
- Option 2: after initial code generation
  - replace the (already generated code of the) subexpression with the temporary.

Proper bookkeeping is necessary to mark the temporaries as alive and still needed.

ECE573, Fall 2005 212

CSE Examples

A = B + C
D = B + C
B = X + Y
E = B + C

A = B + C + D
D = C + D
B = B + C + D
A = B + C

ECE573, Fall 2005 213

CSE: Detecting Equivalent Expressions

- Possible method: create a name for the temporary that is derived from the operators and operands in a unique way.

B = X+Y*Z temp name: X_p_Y_t_Z
X = X+Y-Z temp name: X_p_Y_m_Z
D = X+Y*Z temp name: X_p_Y_t_Z

same temp name : potentially a common subexpression

ECE573, Fall 2005 214

CSE Example with Aliasing Effects

- Do value numbering on the following code :

```
A(i,j)   :=  A(i,j)+B+C;
A(i,j+1) :=  A(i,j)+B+D;
A(i,j)   :=  A(i,j)+B;
```

Basic idea: in addition to doing value numbering on the “pointer”, do value numbering on the “value pointed to” as well.

ECE573, Fall 2005 215

Interprocedural CSE

- So far, we have discussed *intra*-procedural CSE.
- What would it take, to extend the optimization across procedure boundaries?

⇒ Will be discussed later, after Global Data Flow Analysis

ECE573, Fall 2005 216

Factoring Loop-invariant Expressions

- loop-invariant expressions can be moved in front of the loop.
- the idea is similar to common subexpression elimination (reuse computation), however, the action is different:
  - find Def Sets
  - find relevant Variables of expressions (i.e., the variables that are part of the expression)
  - if the two sets are disjoint, it's a loop-invariant expr.

ECE573, Fall 2005 217

Factoring Loop-invariant Expressions (2)

- Array address expressions are important subjects of this optimization, although this is not obvious from the program text

  for example, in A(i,j,k)

  the implicit expression is i*dim(j,k)+j*dim(k)+k

  (assuming row-major storage)
- Safety and profitability is not always guaranteed

  zero-trip loops may never execute the expression

ECE573, Fall 2005 218

Strength Reduction

Replace expensive operations with less-expensive ones. Typically, multiplication is replaced by addition.

Note, the reverse transformation is important too:

→ Finding Induction variables: variables that are incremented by a constant per loop iteration

- examples: the loop variable, statements of the form

  ind = ind + const

  (generalized induction variables: increment can be another induction variable or a multiplication)

ECE573, Fall 2005 219

Strength Reduction Analysis

Induction expression:

I*C + D

Offset
(loop invariant)

Coefficient
(loop invariant)

Loop variable

Note, the induction expressions are different for each loop of a nest

ECE573, Fall 2005 220

Strength Reduction Transformation

- Algorithm :
  - Recognize induction expression, E
  - replace each occurrence of E with temporary T
  - Insert T := I_0 *C+D before loop
    (I_0 is initial value of I in the loop)
  - increment T by C*S at iteration end
    (S is the loop stride)

  (Algorithm on page 642)

ECE573, Fall 2005 221

Global Dataflow Analysis

A general framework for program analysis

222

Global Dataflow Analysis

Analysis of how program attributes change across basic blocks

Dataflow analysis has many diverse applications. A few examples:

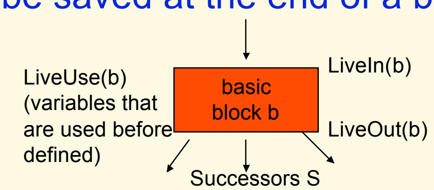
- global live variable analysis
- uninitialized variable analysis
- available expression analysis
- busy expression analysis

(some researchers have suggested to use the term *information propagation* instead of data flow analysis)

ECE573, Fall 2005 223

Live Variable Analysis

Application: the value of dead variables need not be saved at the end of a basic block



LiveUse(b) and Def(b) Are properties of b. They can be analyzed by looking a b only.

$$\mathrm{LiveOut}(b) = \cup\, \mathrm{LiveIn}(i) \quad i \in S$$

$$\mathrm{LiveIn}(b) \supseteq \mathrm{LiveUse}(b)$$
$$\mathrm{LiveIn}(b) \supseteq \mathrm{LiveOut}(b) - \mathrm{Def}(b)$$

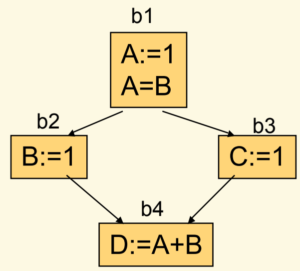
$$\mathrm{LiveIn}(b) = \mathrm{LiveUse}(b) \cup (\mathrm{LiveOut}(b) - \mathrm{Def}(b))$$

This is called a *backward-flow* problem

ECE573, Fall 2005 224

Live Variable Analysis

```
A := 1
if A=B then
  B := 1
else
  C := 1
end if
D := A+B
```



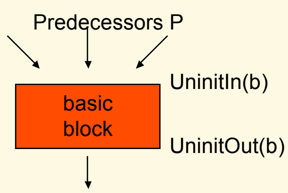
block	Def	LiveUse
b1	{A}	{B}
b2	{B}	∅
b3	{C}	∅
b4	{D}	{A,B}

block	LiveIn	LiveOut
b1	{B}	{A,B}
b2	{A}	{A,B}
b3	{A,B}	{A,B}
b4	{A,B}	∅

ECE573, Fall 2005 225

Uninitialized Variable Analysis

Determine variables that are possibly not initialized



Init(b): variables known to be initialized
Uninit(b): variables that become uninitialized
- assigning "uninit"
- new variables

$$\mathrm{UninitIn}(b) = \bigcup_{i \in P} \mathrm{UninitOut}(i)$$

$$\mathrm{UninitOut}(b) \supseteq \mathrm{UninitIn}(b) - \mathrm{Init}(b)$$
$$\mathrm{UninitOut}(b) \supseteq \mathrm{Uninit}(b)$$

$$\mathrm{UninitOut}(b) = (\mathrm{UninitIn}(b) - \mathrm{Init}(b)) \cup \mathrm{Uninit}(b)$$

This is a *forward-flow* problem

ECE573, Fall 2005 226

Solving Data Flow Equations

- A-cyclic CFG (trivial case):
  - Start at first node, then successively solve equations for successors or predecessors (depending on forward or backward flow problem.)
- Iterative Solutions:
  - Begin with the sets computed locally for each basic block (generically called the Gen and Kill sets)
  - Iterate until the In and Out sets converge

  Is convergence guaranteed?
    - yes, for dataflow graphs with a unique starting node and one or more ending nodes
- Solutions specific to structured languages:

  exploit knowledge about the language constructs that build flow graphs: *if* and *loop* statements

ECE573, Fall 2005 227

Any-Path Flow Problems

- So far, we have considered "any-path" problems: a property holds along *some* path

  In our examples:
  - variable is uninitialized for basic block *b* if it is uninitialized after any predecessor of *b*
  - variable is live if it is used in any successor

ECE573, Fall 2005 228

All-Paths Flow Problems

- All-Paths problems require a property to hold along all possible paths.

Examples:

– Availability of expressions

AvailIn(b) = ∩ AvailOut(i)
AvailOut(b) = Computed(b) ∪ (AvailIn(b)-Killed(b))

– Very-busy expressions

VeryBusyOut(b) = ∩ VeryBusyIn(i)
VeryBusyIn(b) = Used(b) ∪ (VeryBusyOut(b)-Killed(b))

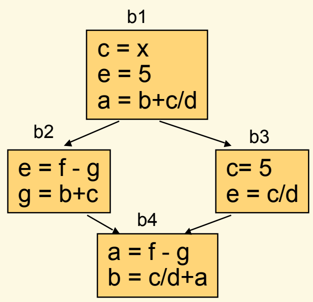
ECE573, Fall 2005 229

General Data Flow Equations

	Forward Flow	Backward Flow
Any Path	Out(b) = Gen(b) ∪ (In(b)-Killed(b)) In(b) = $\bigcup_{i \in P(b)}$ Out(i)	In(b) = Gen(b) ∪ (Out(b)-Killed(b)) Out(b) = $\bigcup_{i \in S(b)}$ In(i)
All Paths	Out(b) = Gen(b) ∪ (In(b)-Killed(b)) In(b) = $\bigcap_{i \in P(b)}$ Out(i)	In(b) = Gen(b) ∪ (Out(b)-Killed(b)) Out(b) = $\bigcap_{i \in S(b)}$ In(i)

ECE573, Fall 2005 230

Exercise: Available Expression Analysis



Notes:

• Precise definition of the meaning of the information to be analyzed is important. Here: an available expression is an expression (given by its relevant variables and operators) whose up-to-date value is available in a temporary variable.

• Note that the value of an available expression from two merged control paths is not necessarily the same. E.g., the value of c/d in b4, depends on the control path taken.

• Be clear about the information sets to be used in the DFA. Here: the sets of all expressions and subexpressions.

ECE573, Fall 2005 231

Other Data Flow Problems and Applications of DF Analyses

- Reaching Definitions (Use-Def Chains)
  - determining use points of assigned values
- Def-Use Chains
  - Can be computed from U-D chains or as a new data flow problem.
- Constant Propagation
  - determining variables that hold constant values.
  - Can be computed based on Reaching Definition information or with an extended data flow analysis
- Copy propagation
  - replacing variables by their assigned expressions and eliminating assignments. A more advanced form of constant propagation.

ECE573, Fall 2005 232

Using the Results of Global Dataflow Analysis:

Global Common Subexpression Elimination

- Do available expression analysis
- Do local CSE
  - Initialize available expressions at the beginning of the basic block with InSet

ECE573, Fall 2005 233

Global Register Allocation

- A possible approach:
  - Do live variable analysis
  - Do local register allocation. Inform successors of the registers holding live variables. Successor initialize their register association lists with this information.
- Issues:
  - Coordinating register use across blocks
  - Deciding where to best place spill code

→ Cooper/Torczon's Global Register Allocation Algorithm

ECE573, Fall 2005 234

Global Register Allocation: Live Ranges

- Live Range:
  - Set of definition and uses, s.t. every definition that may reach a use is in the same live range
  - Some properties of live ranges
    - A variable may have one or several LR
    - LR exist for unnamed variables
    - LR span across basic blocks
    - LR may contain several definitions (LR is simple in BB but complex across multiple BB)

ECE573, Fall 2005 235

Global Register Allocation: Performance Factors

- Inserted spill code
- Inserted copy instructions
- Frequency of execution of basic blocks
  - As a result, definitions, uses, and inserted instructions may execute different numbers of times
- Optimal solution is not feasible (NP hard)

ECE573, Fall 2005 236

Global Register Allocation:
Approach

- Build live ranges
- Assign each live range to a virtual register
  - Rename initially assigned virtual register names
- Annotate instructions (or basic blocks) with their execution frequencies
  - Determine frequencies by static analysis or profiling
- Make decisions:
  - Which LR to reside in registers
  - Which LR to share a register
  - Which specific register for each LR
- Common method used: Graph coloring (use k colors for the nodes of a graph, s.t. adjacent nodes have different colors)

ECE573, Fall 2005 237

Global Register Allocation:
Building Live Ranges

- Build the programs SSA (Static Single Assignment) form.
  - SSA
    - Rename variables s.t. each variable is defined exactly once
    - At control merge points, add a new construct that expresses "variable's value could come from either of the definitions in the merging paths" → $v_9 = \Phi(v_5,v_7)$
- Make a pass through the SSA program, creating sets of variables, s.t., all variables of Φ-function statements are in the same set.
  - Resulting sets are the Live Ranges

ECE573, Fall 2005 238

Global Register Allocation:
Spilling

- Where to spill:
  - Memory hierarchy (cache lines may get evicted if spilled location is not accessed frequently)
  - Special local memory
- Spill cost:
  - Basic cost of memory operation
  - Negative, infinite spill costs for special patterns
  - Multiplied by execution frequencies

ECE573, Fall 2005 239

Global Register Allocation:
Interference Graph and Coloring

- Interference Graph
  - Nodes: Live Ranges
  - Edges: overlaps in Live Ranges

  Algorithm: C&T, Fig 13.7
- Coloring the Interference Graph
  - Coloring is NP-complete → approximate solutions are necessary
  - What to do if we run out of colors (i.e., there are no more registers)
    - (Full) spilling; insert store after each definition; load before each use; reserve some registers for that purpose
    - Splitting the LR: break down the LR into smaller pieces; color the smaller LRs; recombine where necessary

    C&T calls this Top-Down Coloring. There is also Bottom-Up Coloring.
  - Assign priorities to LR. Priority=importance of not spilling the LR. Coloring proceeds in priority order.

ECE573, Fall 2005 240

Other Compiler Transformations

Loop Fusion
Loop Distribution
Loop Interchange
Stripmining (loop blocking)
Tiling

Loop Fusion and Distribution

```
DO j = 1,n
  a(j) = b(j)
ENDDO

DO k=1,n
  c(k) = a(k)
ENDDO
```

fusion →
← distribution

```
DO j = 1,n
  a(j) = b(j)
  c(j) = a(j)
ENDDO
```

- necessary form for vectorization
- can provide synchronization necessary for "forward" dependences
- can create perfectly nested loops

- less parallel loop startup overhead
- can increase *affinity* (better locality of reference)

Both transformations change the statement execution order. Data dependences need to be considered!

Loop Interchange

```
DO i = 1,n
  DO j =1,m
      a(i,j) = b(i,j)
   ENDDO
ENDDO
```

→

```
DO j =1,m
   DO i = 1,n
      a(i,j) = b(i,j)
   ENDDO
ENDDO
```

- loop interchanging alters the data reference order
  → significantly affects locality-of reference
  → data dependences determine the legality of the transformation

- loop interchanging may also impact the granularity of the parallel computation (inner loop may become parallel instead of outer)

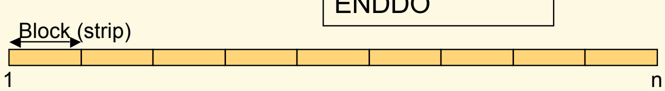
ECE573, Fall 2005 243

Stripmining (a.k.a. Loop Blocking)

```
DO j = 1,n
   a(j) = b(j)
ENDDO
```

→

```
DO j = 1,n,block
   DO k=0,block-1
      a(j+k) = b(j+k)
   ENDDO
ENDDO
```



Many variants:
- adjustment if *n* is not a multiple of *block*
- number of blocks = number of processors
- *cyclic* or *block-cyclic* split

Stripmining can
- Split a loop into two for exploiting hierarchical parallel machines
- Create the right length vectors for vector operations
- Help increase cache locality

ECE573, Fall 2005 244

Tiling

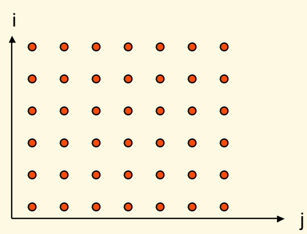
```
DO i = 1,m
  DO j = 1,n
    a(i,j) = a(i,j)+a(i-1,j)
  ENDDO
ENDDO
```

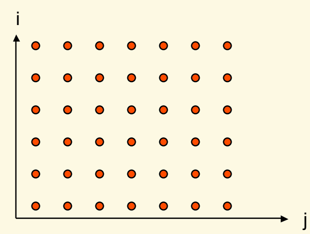
→

```
DO i = 1,m
  DO j = 1,n,block
    DO k=0,block-1
      a(j+k) = …
    ENDDO
  ENDDO
ENDDO
```

→

```
DO j = 1,n,block
  DO i = 1,m
    DO k=0,block-1
      a(j+k) = …
    ENDDO
  ENDDO
ENDDO
```





Analysis and Transformation Techniques for Parallelization

- 1 Data-dependence testing
- 2 Parallelism enabling transformations
- 3 Techniques for multiprocessors

1 Data Dependence Testing

Earlier, we have considered the simple case of a 1-dimensional array enclosed by a single loop:

```
DO i=1,n
  a(4*i) = . . .
  . . .  =  a(2*i+1)
ENDDO
```

the question to answer:
can 4*i ever be equal to 2*I+1 within $i \in [1,n]$?

In general: given
- two subscript functions *f* and *g* and
- loop bounds lower, upper.

Does
$f(i_1) = g(i_2)$ have a solution such that
$lower \leq i_1, i_2 \leq upper$?

ECE573, Fall 2005 247

Data Dependence Tests: Concepts

Terms for data dependences between statements of loop iterations.

- Distance (vector): indicates how many iterations apart are source and sink of dependence.
- Direction (vector): is basically the sign of the distance. There are different notations: (<,=,>) or (-1,0,+1) meaning dependence (from earlier to later, within the same, from later to earlier) iteration.
- Loop-carried (or cross-iteration) dependence and non-loop-carried (or loop-independent) dependence: indicates whether or not a dependence exists within one iteration or across iterations.
  - For detecting parallel loops, only cross-iteration dependences matter.
  - *equal* dependences are relevant for optimizations such as statement reordering and loop distribution.
- Data Dependence Graph: a graph showing statements as nodes and dependences between them as edges. For loops, usually there is only one node per statement instance.
- Iteration Space Graphs: the un-abstracted form of a dependence graph with one node per statement instance. The statements of one loop iteration may be represented as a single node.

ECE573, Fall 2005 248

DDTests: doubly-nested loops

- Multiple loop indices:

```
DO i=1,n
  DO j=1,m
    X(a1*i + b1*j + c1) = . . .
    . . .   =  X(a2*i + b2*j + c2)
  ENDDO
ENDDO
```

dependence problem:

$a_1*i_1 - a_2*i_2 + b_1*j_1 - b_2*j_2 = c_2 - c_1$

$1 \leq i_1, i_2 \leq n$

$1 \leq j_1, j_2 \leq m$

ECE573, Fall 2005 249

DDTests: even more complexity

- Multiple loop indices, multi-dimensional array:

```
DO i=1,n
  DO j=1,m
    X(a1*i1 + b1*j1 + c1, d1*i1 + e1*j1 + f1) = . . .
              . . .   =  X(a2*i2 + b2*j2 + c2, d2*i2 +e2*j2 + f2)
  ENDDO
ENDDO
```

dependence problem:

$a_1*i_1 - a_2*i_2 + b_1*j_1 - b_2*j_2 = c_2 - c_1$

$d_1*i_1 - d_2*i_2 + e_1*j_1 - e_2*j_2 = f_2 - f_1$

$1 \leq i_1, i_2 \leq n$

$1 \leq j_1, j_2 \leq m$

ECE573, Fall 2005 250

Data Dependence Tests:
The Simple Case

Note: variables i_1, i_2 are integers → diophantine equations.

Equation $a * i_1 - b * i_2 = c$ has a solution if and only iff

gcd(a,b) (evenly) divides c

in our example this means: gcd(4,2)=2, which does not divide 1 and thus there is no dependence.

If there **is** a solution, we can test if it lies within the loop bounds. If not, then there is no dependence.

ECE573, Fall 2005 251

Performing the GCD Test

- The diophantine equation

$$a_1{*}i_1 + a_2{*}i_2 + ... + a_n{*}i_n = c$$

has a solution iff $gcd(a_1,a_2,...,a_n)$ evenly divides c

Examples:

15*i +6*j -9*k = 12	has a solution	gcd=3
2*i + 7*j = 3	has a solution	gcd=1
9*i + 3*j + 6*k = 5	has no solution	gcd=3

Euklid Algorithm: find gcd(a,b)

```
Repeat
   a ← a mod b
   swap a,b
Until b=0
```

→*The resulting a is the gcd*

for more than two numbers: gcd(a,b,c) = (gcd(a,gcd(b,c))

ECE573, Fall 2005 252

Other DD Tests

- The GCD test is simple but not accurate
- Other tests
  - Banerjee test: accurate state-of-the-art test
  - Omega test: “precise” test, most accurate for linear subscripts
  - Range test: handles non-linear and symbolic subscripts
  - many variants of these tests

ECE573, Fall 2005 253

The Banerjee(-Wolfe) Test

Basic idea:

if the total subscript range accessed by *ref1* does not overlap with the range accessed by *ref2*, then *ref1* and *ref2* are independent.

```
DO j=1,100
  a(j) = …
  …   = a(j+200)
ENDDO
```

<u>ranges accesses:</u>

[1:100]

[201:300]

→ independent

ECE573, Fall 2005 254

Banerjee(-Wolfe) Test continued

- Weakness of the test:

Consider this dependence

```
DO j=1,100
  a(j) = …
  …  = a(j+5)
ENDDO
```

ranges accesses:
[1:100]
[6:105]
→ independent ?

We did not take into consideration that only loop-carried dependences matter for parallelization.

ECE573, Fall 2005 255

Banerjee(-Wolfe) Test continued

- Solution idea:

for loop-carried dependences factor in the fact that j in *ref2* is greater than in *ref1*

Still considering the potential dependence from a(j) to a(j+5)

```
DO j=1,100
  a(j) = …
  …  = a(j+5)
ENDDO
```

Ranges accessed by iteration j_1 and any other iteration j_2, where $j_1 < j_2$:
[j_1]
[j_1+6:105]
→ Independent for “>” direction

This is commonly referred to as the *Banerjee test with direction vectors.*

Clearly, this loop **has** a dependence. It is an anti-dependence from a(j+5) to a(j)

ECE573, Fall 2005 256

DD Testing with Direction Vectors

Considering direction vectors can increase the complexity of the DD test substantially. For long vectors (corresponding to deeply-nested loops), there are many possible combinations of directions.

$$(d_1,d_2,\ldots,d_n) \longrightarrow \begin{pmatrix} *, & *, & \ldots, & * \\ = & = & & = \\ < & < & & < \\ & > & & > \end{pmatrix}$$

A possible algorithm:

1. try (*,*…*) , i.e., do not consider directions
2. (if not independent) try (<,*,*…*), (=,*,*…*)
3. (if still not independent) try (<,<,*…*),(<,>,*…*) ,(<,=,*…*)
   (=,<,*…*),(=,>,*…*) ,(=,=,*…*)

. . .

(This forms a tree)

ECE573, Fall 2005 257

Non-linear and Symbolic DD Testing

Weakness of most data dependence tests: subscripts and loop bounds must be affine, i.e., linear with integer-constant coefficients

Approach of the Range Test:

capture subscript ranges symbolically

compare ranges: find their upper and lower bounds by determining *monotonicity*. Monotonically increasing/decreasing ranges can be compared by comparing their upper and lower bounds.

ECE573, Fall 2005 258

The Range Test

Basic idea :

1. find the range of array accesses made in a given loop iteration
2. If the upper(lower) bound of this range is less(greater) than the lower(upper) bound of the range accesses in the next iteration, then there is no cross-iteration dependence.

Example: testing independence of the outer loop:

```
DO i=1,n
  DO j=1,m
    A(i*m+j) = 0
  ENDDO
ENDDO
```

range of A accessed in iteration i_x: $[i_x*m+1:\overbrace{(i_x+1)*m}^{ub_x}]$

range of A accessed in iteration i_x+1: $[\underbrace{(i_x+1)*m+1}_{lb_{x+1}}:(i_x+2)*m]$

$ub_x < lb_{x+1} \Rightarrow$ no cross-iteration dependence

ECE573, Fall 2005 259

Range Test continued

we need powerful expression manipulation and comparison utilities

```
DO i1=L1,U1
  ...
  DO in=Ln,Un
    A(f(i0,...in)) = ...
      ...     = A(g(i0,...in))
  ENDDO
 ...
ENDDO
```

Assume *f,g* are monotonically increasing w.r.t. all i_x:
find upper bound of access range at loop k:
successively substitute i_x with U_x, x={n,n-1,...,k}
lowerbound is computed analogously

If *f,g* are monotonically decreasing w.r.t. some i_y, then substitute L_y when computing the upper bound.

we need range analysis

Determining monotonicity: consider d = f(...,i_k,...) - f(...,i_k-1,...)
If d>0 (for all values of i_k) then f is monotonically increasing w.r.t. k
If d<0 (for all values of i_k) then f is monotonically decreasing w.r.t. k

What about symbolic coefficients?
- in many cases they cancel out
- if not, find their range (i.e., all possible values they can assume at this point in the program), and replace them by the upper or lower bound of the range.

ECE573, Fall 2005 260

Range Test : handling non-contiguous ranges

```
DO i1=1,u1
  DO i2=1,u2
     A(n*i1+m*i2)) = …
  ENDDO
ENDDO
```

The basic Range Test finds independence of the outer loop
if n >= u2 and m=1
But not
if n=1 and m>=u1

Idea:
- temporarily (during program analysis) interchange the loops,
- test independence,
- interchange back

Issues:
- legality of loop interchanging,
- change of parallelism as a result of loop interchanging

ECE573, Fall 2005 261

Data-Dependence Test, References

- Banerjee/Wolfe test
  - M.Wolfe, U.Banerjee, "Data Dependence and its Application to Parallel Processing", Int. J. of Parallel Programming, Vol.16, No.2, pp.137-178, 1987
- Range test
  - William Blume and Rudolf Eigenmann. Non-Linear and Symbolic Data Dependence Testing, IEEE Transactions of Parallel and Distributed Systems, Volume 9, Number 12, pages 1180-1194, December 1998.
- Omega test
  - William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence. Proceedings of the 1991 ACM/IEEE Conference on Supercomputing,1991
- I Test
  - Xiangyun Kong, David Klappholz, and Kleanthis Psarris, "The I Test: A New Test for Subscript Data Dependence," *Proceedings of the 1990 International Conference on Parallel Processing*, Vol. II, pages 204-211, August 1990.

ECE573, Fall 2005 262

2 Parallelism Enabling Techniques

Privatization

```
DO i=1,n
  t    = A(i)+B(i)
  C(i) = t + t**2
ENDDO
```

loop-carried anti dependence

scalar privatization

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(t)
DO i=1,n
  t    = A(i)+B(i)
  C(i) = t + t**2
ENDDO
```

```
DO j=1,n
  t(1:m)   = A(j,1:m)+B(j)
  C(j,1:m) = t(1:m) + t(1:m)**2
ENDDO
```

array privatization

```
!$OMP PARALLEL DO
!$OMP+PRIVATE(t)
DO j=1,n
  t(1:m)   = A(j,1:m)+B(j)
  C(j,1:m) = t(1:m) + t(1:m)**2
ENDDO
```

Array Privatization

```
k = 5
DO j=1,n
  t(1:10)   =  A(j,1:10)+B(j)
  C(j,iv) =  t(k)
  t(11:m)   =  A(j,11:m)+B(j)
  C(j,1:m) =  t(1:m)
ENDDO
```

```
DO j=1,n
  IF (cond(j))
     t(1:m)   =  A(j,1:m)+B(j)
     C(j,1:m) =  t(1:m) + t(1:m)**2
  ENDIF
  D(j,1) = t(1)
ENDDO
```

Capabilities needed for Array Privatization

- array Def-Use Analysis
- combining and intersecting subscript ranges
- representing subscript ranges
- representing conditionals under which sections are defined/used
- if ranges too complex to represent: overestimate Uses, underestimate Defs

Array Privatization continued

Array privatization algorithm:

- For each loop nest:
  - iterate from innermost to outermost loop:
    - for each statement in the loop
      - find definitions; add them to the existing definitions in this loop.
      - find array uses; if they are covered by a definition, mark this array section as *privatizable* for this loop, otherwise mark it as upward-exposed in this loop;
    - aggregate defined and upward-exposed, used ranges (expand from range per-iteration to entire iteration space); record them as Defs and Uses for this loop

Array Privatization, References

- Peng Tu and D. Padua. Automatic Array Privatization. Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science 768, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua (Eds.), Springer-Verlag, 1994.
- Zhiyuan Li, Array Privatization for Parallel Execution of Loops, Proceedings of the 1992 ACM International Conference on Supercomputing

Induction Variable Substitution

loop-carried flow dependence

```
ind = k
DO i=1,n
  ind  = ind + 2
  A(ind) = B(i)
ENDDO
```

→

```
Parallel DO i=1,n
  A(k+2*i) = B(i)
ENDDO
```

This is the simple case of an induction variable

Generalized Induction Variables

```
ind=k
DO j=1,n
  ind  = ind + j
  A(ind) = B(j)
ENDDO
```

→

```
Parallel DO j=1,n
  A(k+(j**2+j)/2) = B(i)
ENDDO
```

```
DO i=1,n
  ind1  = ind1 + 1
  ind2  = ind2 + ind1
  A(ind2) = B(i)
ENDDO
```

```
DO i=1,n
  DO j=1,i
    ind  = ind + 1
    A(ind) = B(i)
  ENDDO
ENDDO
```

ECE573, Fall 2005 269

Recognizing GIVs

- Pattern Matching:
  - find induction statements in a loop nest of the form iv=iv+expr or iv=iv*expr, where iv is an scalar integer.
  - expr must be loop-invariant or another induction variable (there must not be cyclic relationships among IVs)
  - iv must not be assigned in a non-induction statement
- Abstract interpretation: find symbolic increments of iv per loop iteration
- SSA-based recognition

ECE573, Fall 2005 270

Computing Closed Form, Substituting additive GIVs

Loop structure L_0: stmt type

```
For j: 1..ub                    
…
S1:  iv=iv+exp        I
…
S2:  loop using iv    L
…
S3:  stmt using iv    U
…
Rof
```

Main:

```
inc = FindIncrement(L0)
Replace(L0,iv)
InsertStatement("iv = inc")   Omit this statement
                              If iv is not live-out
```

For coupled GIVs: begin with independent iv.

Step1: find the increment rel. to start of loop L

```
FindIncrement(L)
  inc=0
  foreach si of type I,L
    if type(si)=L    inc+= FindIncrement(si)
    else             inc += exp
    inc_after[si]=inc
  inc_into_loop[L]= ∑1^(j-1)(inc)  ; inc may depend
  return ∑1^ub(inc)                ; on j
```

Step 2: substitute IV

```
Replace (L,initialval)
  foreach si of type I,L,U
    if type(si)=L     Replace(si,val)
    if type(si)=L,I   val=initialval
                        +inc_into_loop[L]
                        +inc_after[si]
    if type(si)=U   Substitute(si.expr,iv,val)
```

ECE573, Fall 2005 271

Induction Variables, References

- B. Pottenger and R. Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. ACM Int. Conf. on Supercomputing (ICS'95), June 1995. (Extended version: Parallelization in the presence of generalized induction and reduction variables. www.ece.ecn.purdue.edu/~eigenman/reports/1396.pdf)
- Mohammad R. Haghighat , Constantine D. Polychronopoulos, Symbolic analysis for parallelizing compilers, ACM Transactions on Programming Languages and Systems (TOPLAS), v.18 n.4, p.477-518, July 1996
- Michael P. Gerlek , Eric Stoltz , Michael Wolfe, Beyond induction variables: detecting and classifying sequences using a demand-driven SSA form, ACM Transactions on Programming Languages and Systems (TOPLAS), v.17 n.1, p.85-122, Jan. 1995

ECE573, Fall 2005 272

Reduction Parallelization

Scalar Reductions

loop-carried flow dependence

```
DO i=1,n
   sum  = sum + A(i)
ENDDO
```

```
!$OMP PARALLEL PRIVATE(s)
s=0
!$OMP DO
DO i=1,n
    s=s+A(i)
ENDDO
!$OMP ATOMIC
sum = sum+s
!$OMP END PARALLEL
```

```
DO i=1,num_proc
   s(i)=0
ENDDO
!$OMP PARALLEL DO
DO i=1,n
   s(my_proc)=s(my_proc)+A(i)
ENDDO
DO i=1,num_proc
   sum=sum+s(i)
ENDDO
```

Note, OpenMP has a reduction clause, only reduction recognition is needed:

```
!$OMP PARALLEL DO
!$OMP+REDUCTION(+:sum)
DO i=1,n
   sum  = sum + A(i)
ENDDO
```

ECE573, Fall 2

Reduction Parallelization continued

Reduction recognition and parallelization passes:

- induction variable recognition
- **reduction recognition** → **recognizes and annotates reduction variables**
- privatization
- data dependence test
- **reduction parallelization** → **for parallel loops with reduction variables, performance the reduction transformation**

compiler passes

ECE573, Fall 2005 274

Reduction Parallelization

Array Reductions (a.k.a. irregular or histogram reductions)

```
DIMENSION sum(m)
DO i=1,n
   sum(expr)  = sum(expr) + A(i)
ENDDO
```

```
DIMENSION sum(m),s(m)
!$OMP PARALLEL PRIVATE(s)
s(1:m)=0
!$OMP DO
DO i=1,n
    s(expr)=s(expr)+A(i)
ENDDO
!$OMP ATOMIC
sum(1:m) = sum(1:m)+s(1:m)
!$OMP END PARALLEL
```

```
DIMENSION sum(m),s(m,#proc)
!$OMP PARALLEL DO
DO i=1,m
DO j=1,#proc
   s(i,j)=0
ENDDO
ENDDO
!$OMP PARALLEL DO
DO i=1,n
  s(expr,my_proc)=s(expr,my_proc)+A(i)
ENDDO
!$OMP PARALLEL DO
DO i=1,m
DO j=1,#proc
   sum(i)=sum(i)+s(i,j)
ENDDO
ENDDO
```

ECE573, Fall 2005 275

Recognizing Reductions

- Pattern Matching:
  - find reduction statements in a loop of the form
    X=X ⊗ expr ,

    where X is either scalar or an array expression (a[sub], where sub must be the same on the LHS and the RHS),

    ⊗ is a reduction operation, such as +, *, min, max
  - X must not be used in any non-reduction statement in this loop (however, there may be multiple reduction statements for X)

ECE573, Fall 2005 276

Performance Considerations for Reduction Parallelization

- Parallelized reductions execute substantially more code than their serial versions ⇒ overhead if the reduction (*n*) is small.
- In many cases (for large reductions) initialization and sum-up are insignificant.
- False sharing can occur, especially in expanded reductions, if multiple processors use adjacent array elements of the temporary reduction array (*s*).
- Expanded reductions exhibit more parallelism in the sum-up operation.
- Potential overhead in initialization, sum-up, and memory used for large, sparse array reductions ⇒ compression schemes can become useful.

ECE573, Fall 2005 277

Recurrence Substitution

loop-carried flow dependence

```
DO j=1,n
   a(j) = c0+c1*a(j)+c2*a(j-1)+c3*a(j-2)
ENDDO
```

```
call rec_solver(a,n,c0,c1,c2,c3)
```

ECE573, Fall 2005 278

Recurrence Substitution continued

Basic idea of the recurrence solver:

```
DO j=1,40
  a(j) = a(j) + a(j-1)
ENDDO
```

```
DO j=1,10
  a(j) = a(j) + a(j-1)
ENDDO
```

```
DO j=11,20
  a(j) = a(j) + a(j-1)
ENDDO
```

```
DO j=21,30
  a(j) = a(j) + a(j-1)
ENDDO
```

```
DO j=31,40
  a(j) = a(j) + a(j-1)
ENDDO
```

Error: 0   Δa(10)   Δa(10)+Δa(20)   Δa(10)+Δa(20)+Δa(30)

Issues:

- Solver makes several parallel sweeps through the iteration space (n). Overhead can only be amortized if n is large.

- Many variants of the source code are possible. Transformations may be necessary to fit the library call format → additional overhead.

Example from FLO52

```
   DO 40 II=3,IL
   I      = I  -1
   DO 40 J=2,JL
   DW(I,J,N) = DW(I,J,N)  -R*(DW(I,J,N)  -DW(I+1,J,N))
40 CONTINUE
```

ECE573, Fall 2005   279

3 Techniques for Multiprocessors

ECE573, Fall 2005   280

Loop Fusion

```
PARALLEL DO i=1,n
  A(i) = B(i)
ENDDO
PARALLEL DO i=1,n
  C(i) = A(i)+D(i)
ENDDO
```

loop fusion →

```
PARALLEL DO i=1,n
  A(i) = B(i)
  C(i) = A(i)+D(i)
ENDDO
```

Loop fusion is the reverse of loop distribution.
It reduces the loop fork/join overhead.

Loop Coalescing

```
PARALLEL DO i=1,n
  DO j=1,m
    A(i,j) = B(i,j)
  ENDDO
ENDDO
```

loop coalescing →

```
PARALLEL DO ij=1,n*m
   i = 1 + (ij-1) DIV m
   j = 1 + (ij-1) MOD m
  A(i,j) = B(i,j)
ENDDO
```

Loop coalescing
- can increase the number of iterations of a parallel loop → load balancing
- adds additional computation → overhead

Loop Interchange

```
DO i=1,n
  PARALLEL DO j=1,m
    A(i,j) = A(i-1,j)
  ENDDO
ENDDO
```

loop interchange →

```
PARALLEL DO j=1,m
  DO i=1,n
    A(i,j) = A(i-1,j)
  ENDDO
ENDDO
```

Loop interchange affects:

- granularity of parallel computation (compare the number of parallel loops started)
- locality of reference (compare the cache-line reuse)

these two effects may impact the performance in the same or in opposite directions.

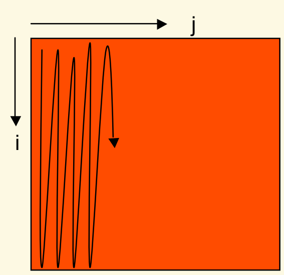
ECE573, Fall 2005 283

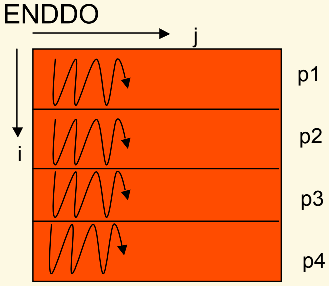
Loop Blocking

```
DO j=1,m
  DO i=1,n
    B(i,j)=A(i,j)+A(i,j-1)
  ENDDO
ENDDO
```

loop blocking →

```
DO PARALLEL i1=1,n,block
  DO j=1,m
    DO i=i1,min(i1+block-1,n)
      B(i,j)=A(i,j)+A(i,j-1)
    ENDDO
  ENDDO
ENDDO
```





This is basically the same transformation as stripming.
However, loop interchanging is involved as well.

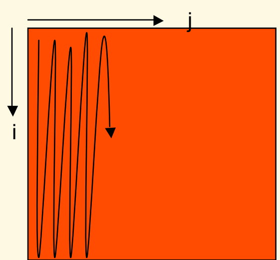
ECE573, Fall 2005 284

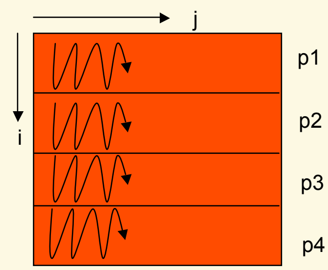
Loop Blocking
continued

```
DO j=1,m
  DO i=1,n
    B(i,j)=A(i,j)+A(i,j-1)
  ENDDO
ENDDO
```

→

```
!$OMP PARALLEL
DO j=1,m
!$OMP DO
  DO i=1,n
    B(i,j)=A(i,j)+A(i,j-1)
  ENDDO
!$OMP ENDDO NOWAIT
ENDDO
!$OMP END PARALLEL
```





Choosing the Block Size

The block size must be small enough so that all data references between the use and the reuse fit in cache.

```
DO j=1,m
  DO k=1,block
    … (r1 data references)
    … = A(k,j) + A(k,j-d)
    … (r2 data references)
  ENDDO
ENDDO
```

Number of references made between the access A(k,j) and the access A(k,j-d) when referencing the same memory location: (r1+r2+3)*d*block

→ **block < cachesize / (r1+r2+2)*d**

If the cache is shared, all processors use it simultaneously. Hence the effective cache size appears smaller:

block < cachesize / (r1+r2+2)*d*num_proc

Loop Distribution Enables Other Techniques

```
DO i=1,n
  A(i) = B(i)
  DO j=1,m
    D(i,j)=E(i,j)
  ENDDO
ENDDO
```

loop distribution enables interchange →

```
DO i=1,n
  A(i) = B(i)
ENDDO

DO j=1,m
  DO i=1,n
    D(i,j)=E(i,j)
  ENDDO
ENDDO
```

In a program with multiply-nested loops, there can be a large number of possible program variants obtained through distribution and interchanging

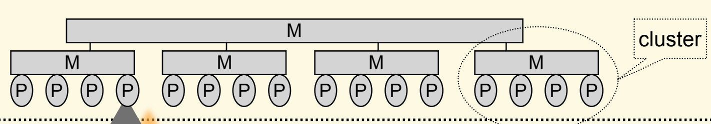
ECE573, Fall 2005 287

Multi-level Parallelism from Single Loops

```
DO i=1,n
 A(i) = B(i)
ENDDO
```

↓ strip mining for multi-level parallelism

PARALLEL DO *(inter-cluster)* i1=1,n,strip
  PARALLEL DO *(intra-cluster)* i=i1,min(i1+strip-1,n)
    A(i) = B(i)
  ENDDO
ENDDO



ECE573, Fall 2005 288

References

- **High Performance Compilers for Parallel Computing**, Michale Wolfe, Addison-Wesley, ISBN 0-8053-2730-4.
- **Optimizing Compilers for Modern Architectures**: A Dependence-based Approach, Ken Kennedy and John R. Allen, Morgan Kaufmann Publishers, ISBN 1558602860