RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Code Generation

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

- Homework 8 will be posted tomorrow
- Extra-credit project will be announced this Thursday
  - → Extra-credit project can replace the third project (+ 10% bonus)
  - → Extra-credit is a lot harder than expected, so please sign up if you are interested and if you are up for a challenge
  - → A global register allocator for real GPU programs
    - You have to do a lot of reading and self-studying
    - Learn GPU instruction set architecture (ISA)
    - Need to perform data flow analysis
    - Need to handle procedures
    - Need to handle multi-class registers
    - Need to think about trade-off between concurrency & register allocation
  - → But it will be really exciting!
  - → Lectures for last two weeks will be on global register allocation

Roadmap

- Review on error detection/recovery
- Review on type expression
- Code Generation
  - → Expression
  - → Handling Assignment
  - → Array
  - → Boolean and Relational Values
  - → Control Flow

Error Recovery in Shift-Reduce Parsers

The problem: parser encounters an invalid token

Goal: Want to parse the rest of the file

Basic idea:

- Assume something went wrong while trying to find handle for nonterminal A
- Pretend handle for A has been found; pop "handle", skip over input to find terminal that can follow A

Restarting the parser:

- find a restartable state on the stack (has transition for nonterminal A)
- move to a consistent place in the input (token that can follow A)
- perform (error) reduction (for nonterminal A)
- print an informative message

Review: Error Recovery in YACC

Yacc's (bison's) error mechanism (note: version dependent!)

- designated token **`error`**
- used in error productions of the form

  $A \rightarrow$ **`error`** α // basic case
- α specifies synchronization points

When error is discovered

- pops stack until it finds state where it can shift the **`error`** token
- resumes parsing to match α

  special cases:
  - α = w, where w is string of terminals: skip input until w has been read
  - $\alpha = \varepsilon$: skip input until state transition on input token is defined
- error productions can have actions

Review: Error Recovery in YACC

```
cmpdstmt: BEG stmt_list END
stmt_list : stmt
          | stmt_list  ';'  stmt
          | error { yyerror("\n***Error: illegal statement\n");}
```

This should

- throw out the erroneous statement
- synchronize at ";" or "END" (implicit: $\alpha = \varepsilon$)
- writes message "***Error: illegal statement" to stderr

Example: begin a & 5 | hello ; a := 3 end

↑ ↑ resume parsing

***Error: illegal statement

Today's topics

- Review on error detection/recovery
- Review on type expression
- Code Generation
  - → Expression
  - → Handling Assignment
  - → Array
  - → Boolean and Relational Values
  - → Control Flow

Review - Types and Type Systems

Type system: Each language construct (operator, expression, statement, …) is associated with a type expression. The type system is a collection of rules for assigning type expressions to these constructs.

Type expressions for

- basic types: `integer, char, real, boolean, typeError`
- constructed types, e.g., one-dimensional arrays:
  `array(lb, ub,` elem_type`)` , where elem_type is a type expression

A type checker implements a type system. It computes or "constructs" type expressions for each language construct

Types and Type Systems

Example type inference rule:

$$\frac{E \vdash e_1 : \text{integer} \quad E \vdash e_2 : \text{integer}}{E \vdash (e_1 + e_2) : \text{integer}}$$

where E is a type environment that maps constants and variables to their type expressions.

Questions: How to specify rules that allow type coercion (type widening) from integers to reals in arithmetic expressions?

3.0 + 1 or 1 + 3.0

Types and Type Systems

Example type inference rule:

$$\frac{E \vdash e_1 : \text{integer} \quad E \vdash e_2 : \text{integer}}{E \vdash (e_1 + e_2) : \text{integer}}$$

where E is a type environment that maps constants and variables to their type expressions.

Questions: How to specify rules that allow type coercion (type widening) from integers to reals in arithmetic expressions?

3.0 + 1 or 1 + 3.0

Types and Type Systems

Example type inference rule pointer dereferencing:

$$\frac{E \vdash e : ???}{E \vdash *e : ???}$$

where E is a type environment that maps constants and variables to their type expressions.

Types and Type Systems

Example type inference rule pointer dereferencing:

$$\frac{E \vdash e : \text{pointer(integer)}}{E \vdash *e : \text{integer}}$$

where E is a type environment that maps constants and variables to their type expressions.

pointer(…) is now part of the type expression language such as array(…).

Types and Type Systems

Example type inference rule pointer dereferencing:

$$\frac{E \vdash e : \text{pointer}(\beta)}{E \vdash *e : \beta}$$

where E is a type environment that maps constants and variables to their type expressions.

Type expressions may also contain type variables such as β. Type variables can denote any type expression.

Type variables are needed to express polymorphic types.

Types and Type Systems

Example type inference rule address computation:

$$\frac{E \vdash e : \text{integer}}{E \vdash \&e : ???}$$

where E is a type environment that maps constants and variables to their type expressions.

Types and Type Systems

Example type inference rule address computation:

$$\frac{E \vdash e : \text{integer}}{E \vdash \&e : \text{pointer(integer)}}$$

where E is a type environment that maps constants and variables to their type expressions.

Types and Type Systems

Formal proof that a program can be typed correctly.

```
int a;
. . .
. . . *(&a) + 3 . . .
```

Types and Type Systems

Formal proof that a program can be typed correctly.

```
int a;
. . .
. . . *(&a) + 3 . . .
```

$$\frac{E \vdash e : \text{pointer}(\beta)}{E \vdash *e : \beta}$$

$$\frac{E \vdash e : \text{integer}}{E \vdash \&e : \text{pointer(integer)}}$$

Types and Type Systems

Programmers may define their own types and give them names:

```
type my_int is int;
 ...
int a;
my_int b;
 ...
  ... a + b ...
```

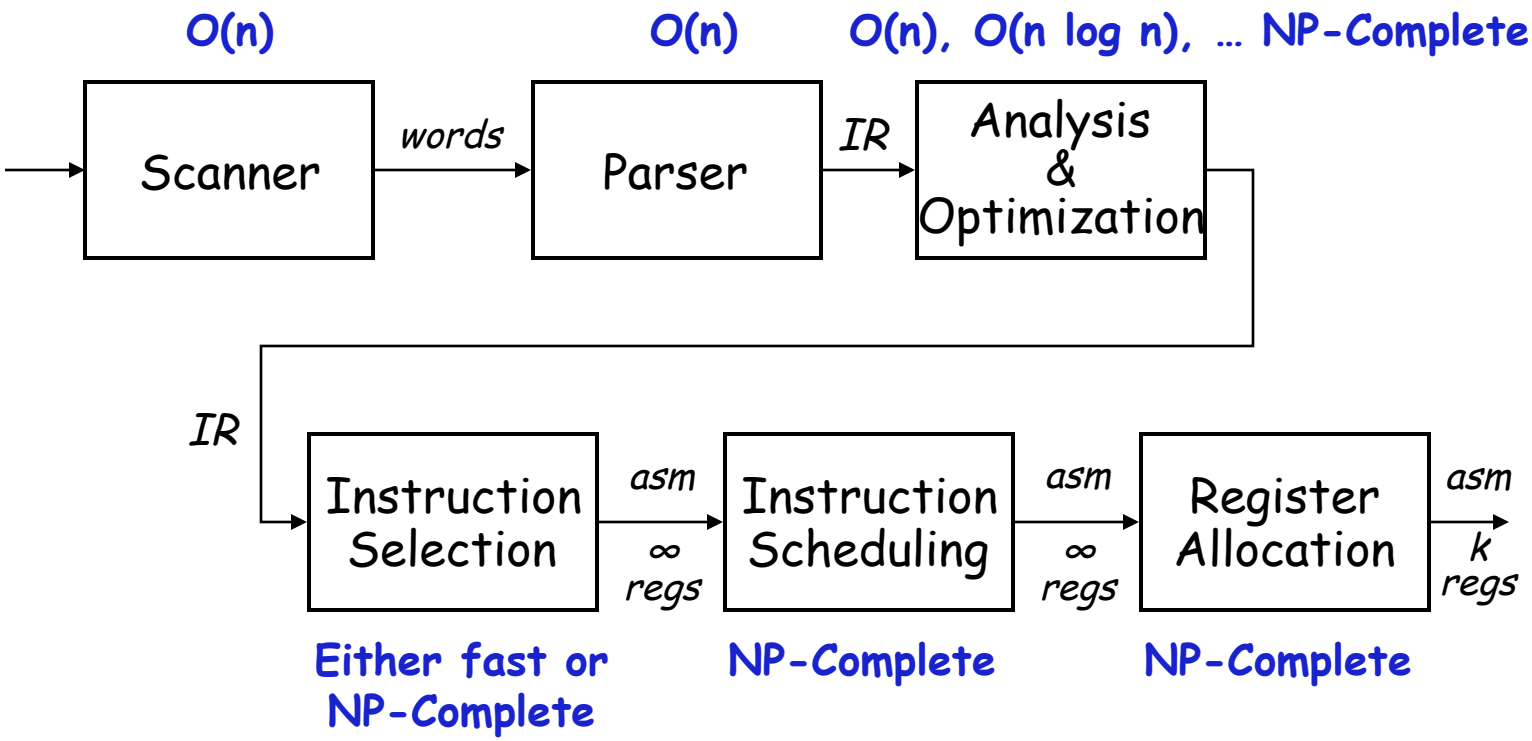
Type names can also be part of the type expression language.
Note: type names and type variables are different!

Roadmap

- Review on error detection/recovery
- Review on type expression
- Code Generation
  - → Expression
  - → Handling Assignment
  - → Array
  - → Boolean and Relational Values
  - → Control Flow

Code Generation: Structure of a Compiler

(EaC Ch. 7)



A compiler is a lot of fast stuff followed by some hard problems

- → The hard stuff is mostly in code generation and optimization
- → For superscalars, its allocation & scheduling that is particularly important

Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,- :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
       }
       return result;
 }
```

"Expr" returns virtual register number that will contain result of subtree evaluation at runtime

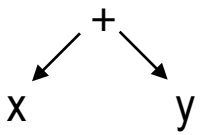
The concept

- **Use a simple treewalk evaluator**
- **Bury complexity in routines it calls**
  - > *base*(), *offset*(), & *val*()
- **Implements expected behavior**
  - > **Visits & evaluates children**
  - > **Emits code for the op itself**
  - > **Returns register with result**
- **Works for simple expressions**
- **Easily extended to other operators**
- **Does not handle control flow**

Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,- :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
       }
       return result;
 }
```

Example:



Produces:

expr("x") →

```
loadI    @x      ⇒ r1
loadAO   r0, r1  ⇒ r2
```

expr("y") →

```
loadI    @y      ⇒ r3
loadAO   r0, r3  ⇒ r4
```

NextRegister() → r5

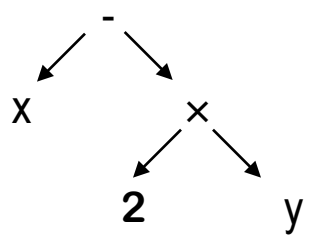
emit(add,r2,r4,r5) →

```
add      r2, r4  ⇒ r5
```

Generating Code for Expressions

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,- :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
       }
       return result;
 }
```

Example:



Generates:

loadI	**@x**	**⇒ r1**
loadAO	**r0, r1**	**⇒ r2**
loadI	**2**	**⇒ r3**
loadI	**@y**	**⇒ r4**
loadAO	**r0,r4**	**⇒ r5**
mult	**r3, r5**	**⇒ r6**
sub	**r2, r6**	**⇒ r7**

Extending the Simple Treewalk Algorithm

More complex cases for IDENTIFIER

- What about values in registers?
  - → Modify the **IDENTIFIER** case
  - → Already in a register ⇒ return the register name
  - → Not in a register ⇒ load it as before, but record the fact
  - → Choose names to avoid creating false dependences ("fresh" virtual register)
- What about parameter values?
  - → Many linkages pass the first several values in registers
  - → Call-by-value ⇒ just a local variable with "funny" offset
  - → Call-by-reference ⇒ needs an extra indirection
- What about function calls in expressions?
  - → Generate the calling sequence & load the return value
  - → Might limits compiler's ability to reorder operations

Extending the Simple Treewalk Algorithm

Adding other operators (in addition to +, -, *, /)

- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls
- Handle assignment as an operator

Mixed-type expressions

- Insert conversions as needed from conversion table
- Most languages have symmetric & rational conversion tables

Typical Addition Table

+	Integer	Real	Double	Complex
Integer	Integer	Real	Double	Complex
Real	Real	Real	Double	Complex
Double	Double	Double	Double	Complex
Complex	Complex	Complex	Complex	Complex

Extending the Simple Treewalk Algorithm

What about evaluation order?

- Can use commutativity & associativity to improve code
- This problem is truly hard

What about order of evaluating operands?

- 1st operand must be preserved while 2nd is evaluated
- Takes an extra register for 2nd operand
- Should evaluate more demanding operand expression first

Taken to its logical conclusion, this creates Sethi-Ullman register allocation scheme (p. 570-571 in ALSU)

Generating Code in the Parser

Need to generate an initial IR form

- Might generate an AST, use it for some high-level, near-source work (type checking, optimization), then traverse it and emit a lower-level IR similar to **ILOC**

The big picture

- Recursive *expr(node)* algorithm really works bottom-up
  - → Actions on non-leaves occur after children are done
- Can encode same basic structure into *ad-hoc* SDT scheme
  - → Identifiers load themselves & stack (live in parser stack) virtual register name
  - → Operators emit appropriate code & stack resulting VR name
  - → Assignment requires evaluation to an *lvalue* or an *rvalue*

Ad-hoc SDT versus a Recursive Treewalk

```
expr(node) {
  int result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,- :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2← offset(node);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
       }
       return result;
 }
```

```
Goal :    Expr  { $$ = $1; } ;
Expr:     Expr PLUS Term
          { t = NextRegister();
            emit(add,$1,$3,t); $$ = t; }
      |   Expr MINUS Term  {…}
      |   Term { $$ = $1; } ;
Term:     Term TIMES Factor
          { t = NextRegister();
            emit(mult,$1,$3,t); $$ = t; };
      |   Term DIVIDES Factor {…}
      |   Factor { $$ = $1; };
Factor:   NUMBER
          { t = NextRegister();
            emit(loadI,val($1),none, t );
            $$ = t; }
      |   ID
           { t1 = base($1);
             t2 = offset($1);
             t = NextRegister();
            emit(loadAO,t1,t2,t);
            $$ =  t; }
```

Roadmap

- Review on error detection/recovery
- Review on type expression
- Code Generation
  - → Expression
  - → Handling Assignment
  - → Array
  - → Boolean and Relational Values
  - → Control Flow

Handling Assignment (just another operator)

lhs ← *rhs*

Strategy

- Evaluate *rhs* to a value *(an rvalue)*
- Evaluate *lhs* to a location *(an lvalue)*
  - → *lvalue* is a register ⇒ move rhs
  - → *lvalue* is an address ⇒ store rhs
- If *rvalue* & *lvalue* have different types
  - → Evaluate *rvalue* to its "*natural*" type
  - → Convert that value to the type of **lvalue*

Unambiguous scalars may go into registers (no aliasing)

Ambiguous scalars or aggregates go into memory (possible aliasing)

Handling Assignment

What if the compiler cannot determine the rhs's type ?

- This is a property of the language & the specific program
- If type-safety is desired, compiler must insert a run-time check
- Add a *tag field* to the data items to hold type information

Code for assignment becomes more complex

```
evaluate rhs
if type(lhs) ≠ rhs.tag
   then
      convert rhs to type(lhs) or
      signal a run-time error
lhs ← rhs
```

This is much more complex than if it knew the types

Handling Assignment

Compile-time type-checking

- Goal is to eliminate both the check & the tag
- Determine, at compile time, the type of each subexpression
- Use compile-time types to determine if a run-time check is needed

Optimization strategy

- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation (superscalar or multi-core architectures)

Handling Assignment (with reference counting)

The problem with reference counting

- Must adjust the count on each pointer assignment
- Overhead is significant, relative to assignment

Code for assignment becomes

```
evaluate rhs
lhs→count ← lhs→count - 1
lhs ← addr(rhs)
rhs→count ← rhs→count + 1
```

Plus a check for zero at the end

This adds *1 +, 1 -, 2 loads, & 2 stores*

With extra functional units & large caches, this may become either cheap or free. What about power consumption?

Next class

More code generation

Read EaC: Chapter 7.1 – 7.5