Undergraduate Compilers Review

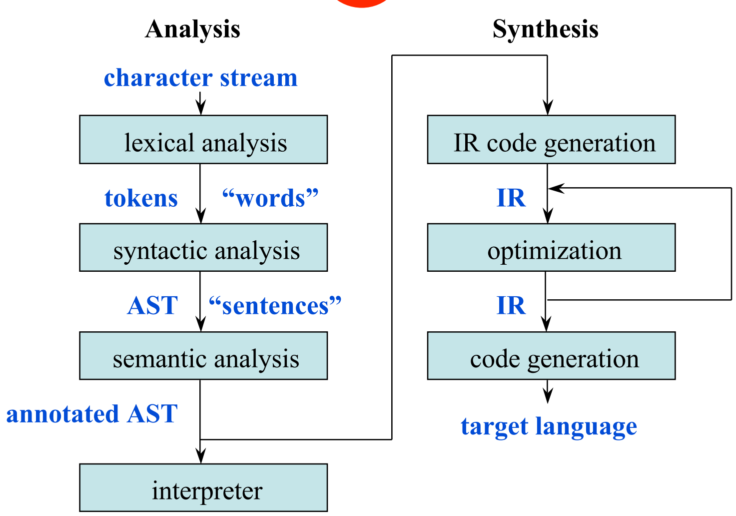
Announcements

- Makeup lectures on Aug 29th and Sept 9th

Today

- Overall structure of a compiler
- OpenAnalysis
- Intermediate representations

Structure of a Typical ~~Interpreter~~ Compiler



Lexical Analysis (Scanning)

Break character stream into tokens ("words")

– Tokens, lexemes, and patterns

– Lexical analyzers are usually automatically generated from patterns (regular expressions) (*e.g.,* lex)

Examples

<table>
<tr><th>token</th><th>lexeme(s)</th><th>pattern</th></tr>
<tr><td>const</td><td><code>const</code></td><td><code>const</code></td></tr>
<tr><td>if</td><td><code>if</code></td><td><code>if</code></td></tr>
<tr><td>relation</td><td><code><,<=,=,!=,...</code></td><td><code>< | <= | = | != | ...</code></td></tr>
<tr><td>identifier</td><td><code>foo,index</code></td><td><code>[a-zA-Z_]+[a-zA-Z0-9_]*</code></td></tr>
<tr><td>number</td><td><code>3.14159,570</code></td><td><code>[0-9]+ | [0-9]*.[0-9]+</code></td></tr>
<tr><td>string</td><td><code>"hi", "mom"</code></td><td><code>".*"</code></td></tr>
</table>

`const pi := 3.14159` ⇒ *const, identifier*(`pi`), *assign,number*(**3.14159**)

Syntactic Analysis (Parsing)

Impose structure on token stream

– Limited to syntactic structure (⇒ high-level)

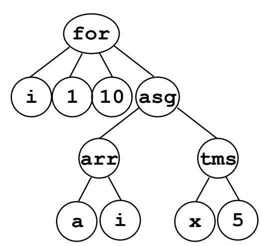
– Parsers are usually automatically generated from grammars (*e.g.,* yacc, bison, cup, javacc), which use shift-reduce parsing

– An implicit parse tree occurs during parsing as grammer rules are matched

– Output of parsing is usually represented with an *abstract syntax tree* (AST)

Example

```
for i = 1 to 10 do
   a[i] = x * 5;
```



for id(i) *equal number*(1) *to number*(10) *do*
id(a) *lbracket id*(i) *rbracket equal id*(x) *times number*(5) *semi*

Bottom-Up Parsing: Shift-Reduce

Grammer

```
(1) S -> E
(2) E -> E + T
(3) E -> T
(4) T -> id
```

a + b + c

```
S -> E
  -> E + T
  -> E + id
  -> E + T + id
  -> E + id + id
  -> T + id + id
  -> id + id + id
```

Rightmost derivation: expand rightmost non-terminals first

Yacc and bison generate shift-reduce parsers:

- LALR(1): look-ahead, left-to-right, rightmost derivation in reverse, 1 symbol lookahead
- LALR is a parsing table construction method, smaller tables than canonical LR

Reference: Barbara Ryder's 198:515 lecture notes

Shift-Reduce Parsing Example

Stack	Input	Action
$	a + b + c	shift
$ a	+ b + c	reduce (4)
$ T	+ b + c	reduce (3)
$ E	+ b + c	shift
$ E +	b + c	shift
$ E + b	+ c	reduce (4)
$ E + T	+ c	reduce (2)
$ E	+ c	shift
$ E +	c	shift
$ E + c		reduce (4)
$ E + T		reduce (2)
$ E		reduce (1)
$ S		accept

```
(1) S -> E
(2) E -> E + T
(3) E -> T
(4) T -> id
```

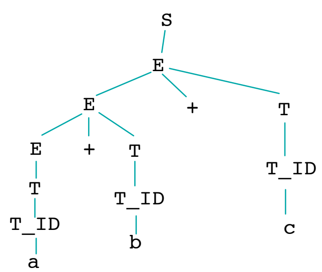
Reference: Barbara Ryder's 198:515 lecture notes

Syntax-directed Translation: AST Construction example

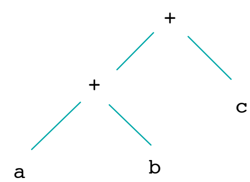
Grammer with production rules

```
S:  E        { $$ = $1; };
E:  E '+' T  { $$ = new node("+", $1, $3); }
  | T        { $$ = $1; }
  ;
T: T_ID      { $$ = new leaf("id", $1); };
```

Implicit parse tree for a+b+c



AST for a+b+c



Reference: Barbara Ryder's 198:515 lecture notes

Project 1: Basic Outline

1) **Download and build OpenAnalysis**
2) **Copy Project1.tar to your CS directory and build**
3) **Implement 3 parsers that build up certain parts of a subsidiary IR using the examples in testSubIR.cpp and Input/testSubIR.oa**
4) **Next week start testing FIAlias implementation in OpenAnalysis**

OpenAnalysis

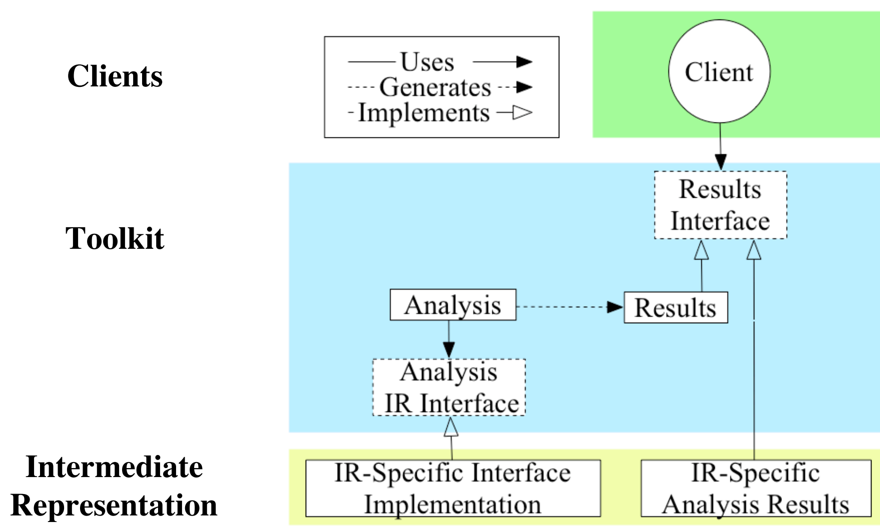
Problem: Insufficient analysis support in existing compiler infrastructures due to non-transferability of analysis implementations

Decouples analysis algorithms from intermediate representations (IRs) by developing analysis-specific interfaces

Analysis reuse across compiler infrastructures

- Enable researchers to leverage prior work
- Enable direct comparisons amongst analyses
- Increase the impact of compiler analysis research

Software Architecture for OpenAnalysis



Project 1: Scanners and Parsers for OpenAnalysis Test Input

```
// int main() {
       PROCEDURE = { < ProcHandle("main"), SymHandle("main") > }

           // int x;
           LOCATION = { < SymHandle("x"), local > }
           // int *p;
           LOCATION = { < SymHandle("p"), local > }

           // all other symbols visible to this procedure
           LOCATION = { < SymHandle("g"), not local > }

           // x = g;
           MEMREFEXPRS = { StmtHandle("x = g;") =>
               [
                   MemRefHandle("x_1") => NamedRef(DEF, SymHandle("x") )
                   MemRefHandle("g_1") => NamedRef(USE, SymHandle("g") )
               ] }
```


Project Hints

testSubIR.cpp has calls that your parsers must execute when it parses testSubIR.oa

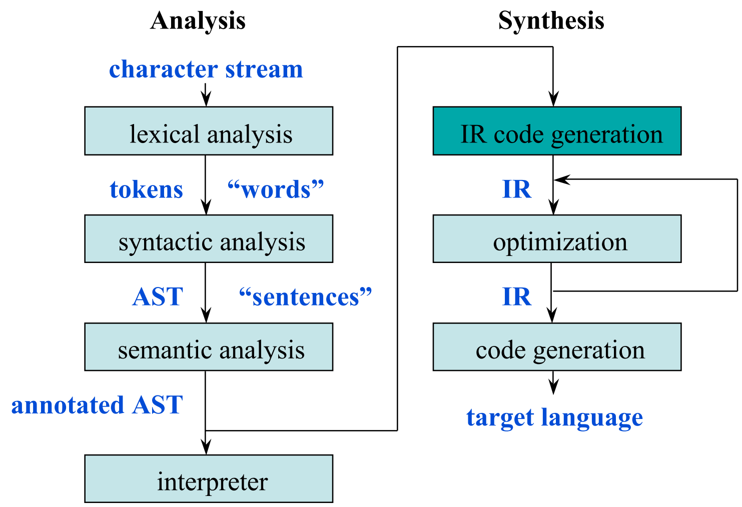
Assume correct input

Sending lists up the parse tree

```
SymList:
        SymList Sym
        {
            $1->push_back(*$2);
            $$ = $1;
            delete $2;
        }
    |   /* empty */
        {
            $$ = new std::list<OA::SymHandle>;
        }
    ;
```

Typo in writeup: "uncomment" parts of testSubIR.oa as you create each parser

Structure of a Typical Compiler



Semantic Analysis

Determine whether source is meaningful

- Check for semantic errors
- Check for type errors
- Gather type information for subsequent stages
  - Relate variable uses to their declarations
- Some semantic analysis takes place during parsing

Example errors (from C)

```
function1 = 3.14159;
x = 570 + "hello, world!"
scalar[i]
```

Compiler Data Structures

Symbol Tables

- Compile-time data structure
- Holds names, type information, and *scope* information for variables

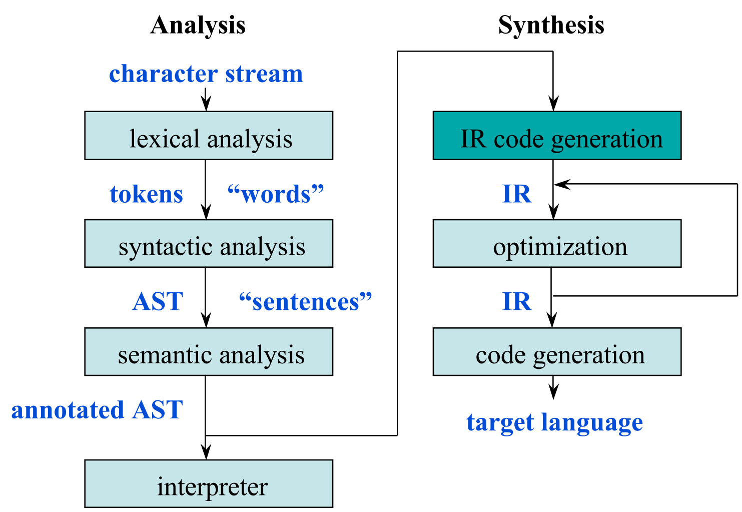
Scopes

- A name space
  e.g., In Pascal, each procedure creates a new scope
  e.g., In C, each set of curly braces defines a new scope
- Can create a separate symbol table for each scope

Using Symbol Tables

- For each variable declaration:
  - Check for symbol table entry
  - Add new entry (parsing); add type info (semantic analysis)
- For each variable use:
  - Check symbol table entry (semantic analysis)

Structure of a Typical Compiler



IR Code Generation

Goal

– Transforms AST into low-level *intermediate representation* (IR)

Simplifies the IR

– Removes high-level control structures: **`for`**, **`while`**, **`do`**, **`switch`**

– Removes high-level data structures: arrays, structs, unions, enums

Results in assembly-like code

– Semantic lowering

– Control-flow expressed in terms of "gotos"

– Each expression is very simple (three-address code)

e.g., `x := a * b * c` ⇒ `t := a * b`
`x := t * c`

A Low-Level IR

Register Transfer Language (RTL)

– Linear representation

– Typically language-independent

– Nearly corresponds to machine instructions

Example operations

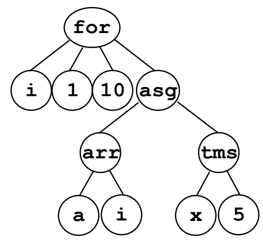
– Assignment	`x := y`
– Unary op	`x := op y`
– Binary op	`x := y op z`
– Address of	`p := & y`
– Load	`x := *(p+4)`
– Store	`*(p+4) := y`
– Call	`x := f()`
– Branch	`goto L1`
– Cbranch	`if (x==3) goto L1`

Example

Source code

```
for i = 1 to 10 do
  a[i] = x * 5;
```

High-level IR (AST)



Low-level IR (RTL)

```
        i := 1
loop1:
        t1 := x * 5
        t2 := &a
        t3 := sizeof(int)
        t4 := t3 * i
        t5 := t2 + t4
        *t5 := t1
        i := i + 1
        if i <= 10 goto loop1
```


Compiling Control Flow

Switch statements

– Convert **`switch`** into low-level IR

e.g.,

```
switch (c) {
    case 0: f();
            break;
    case 1: g();
            break;
    case 2: h();
            break;
}
```

```
        if (c!=0) goto next1
        f ()
        goto done
next1:  if (c!=1) goto next2
        g()
        goto done
next2:  if (c!=3) goto done
        h()
done:
```

– Optimizations (depending on size and density of cases)
  – Create a jump table (store branch targets in table)
  – Use binary search

Compiling Arrays

Array declaration

– Store name, size, and type in symbol table

Array allocation

– Call **`malloc()`** or create space on the runtime stack

Array referencing

– *e.g.*, **`A[i]`** ⇒ **`*(&A + i * sizeof(A_elem))`**

```
t1 := &A
t2 := sizeof(A_elem)
t3 := i * t2
t4 := t1 + t3
*t4
```


Compiling Procedures

Properties of procedures

– Procedures define scopes

– Procedure lifetimes are nested

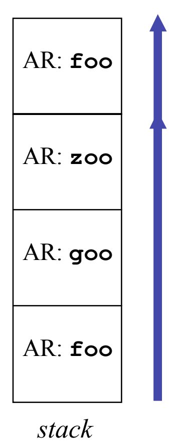
– Can store information related to dynamic invocation of a procedure on a call stack (*activation record* or AR or stack frame):

  – Space for saving registers

  – Space for passing parameters and returning values

  – Space for local variables

  – Return address of calling instruction



Stack management

– Push an AR on procedure entry

– Pop an AR on procedure exit

– Why do we need a stack?

Compiling Procedures (cont)

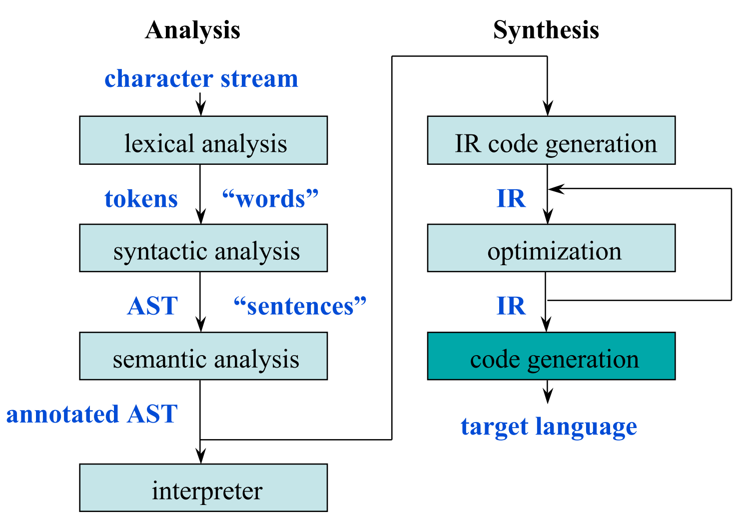
Code generation for procedures

- Emit code to manage the stack
- Are we done?

Translate procedure body

- References to local variables must be translated to refer to the current activation record
- References to non-local variables must be translated to refer to the appropriate activation record or global data space

Structure of a Typical Compiler



Code Generation

Conceptually easy

- Three address code is a generic machine language
- Instruction selection converts the low-level IR to real machine instructions

The source of heroic effort on modern architectures

- Alias analysis
- Instruction scheduling for ILP
- Register allocation
- More later. . .

Concepts

Compilation stages

- Scanning, parsing, semantic analysis, intermediate code generation, optimization, code generation

Representations

- AST, low-level IR (RTL)

Next Time

Reading

– Chapter 8.1 in Muchnick

Lecture

– Finish Undergrad Compilers Review
– Dataflow analysis

Language Implementation Timeline

For entertainment purposes only!