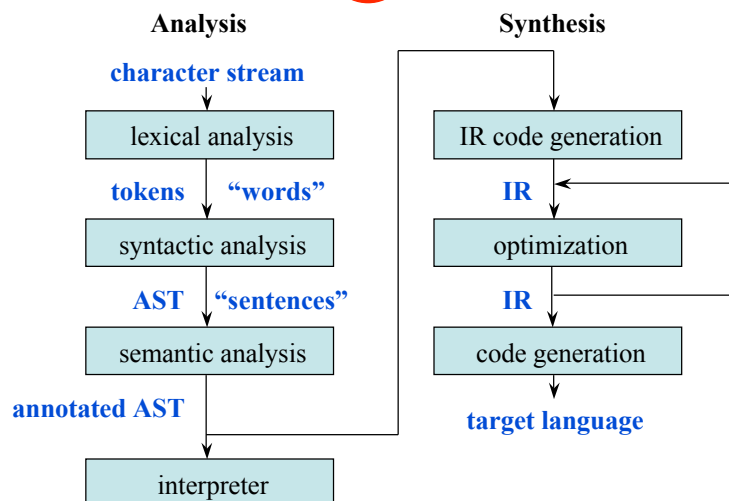


Undergraduate Compilers in a Day

Today

- Overall structure of a compiler
- 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, flex)

Examples

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

const pi := 3.14159 \Rightarrow *const, identifier(pi), assign, number(3.14159)*

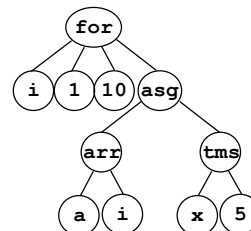
Syntactic Analysis (Parsing)

Impose structure on token stream

- Limited to syntactic structure
- Structure usually represented with an *abstract syntax tree* (AST)
- Theory meets practice:
 - Regular expressions, formal languages, grammars, parsing...
- Parsers are usually automatically generated from grammars (e.g., yacc, bison, cup, javacc)

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*

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 = 2.718282;  
x = 570 + "hello, world!"  
scalar[i]
```

Compiler Data Structures

Symbol Tables

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

Scope

- 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 with type info
- For each variable use:
 - Check symbol table entry

Symbol Table Alternative

Idea

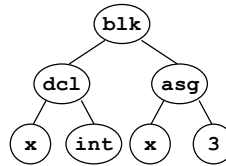
- Dispense with explicit symbol table structure
- Include declarations in AST

Why?

- Source language syntax matches access structure (in C, scoping is mostly flat and data types are primitive)
- Simple
- Easy to generate C code (with declarations)

Example

```
{  
  int x;  
  x = 3;  
}
```

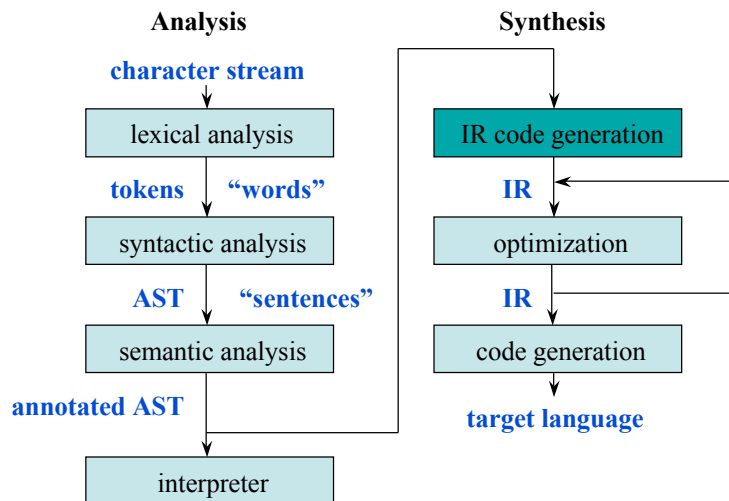


CIS570 Lecture 2

Undergraduate Compilers in a Day

8

Structure of a Typical Compiler



CIS570 Lecture 2

Undergraduate Compilers in a Day

9

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 $x := *(p+c)$
- Store $*(p+c) := y$
- Call $x := f()$
- Branch **goto** L1
- Cbranch **if** (x==3) **goto** L1

Example

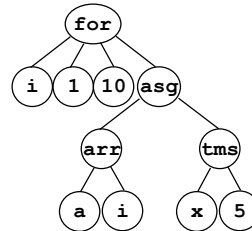
Source code

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

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
```

High-level IR (AST)



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 Control Flow (cont)

Switch statements (cont)

- Convert **switch** into optimized (jump table) low-level IR

```
e.g., switch (c) {           jtarr: .words targ0,targ1,targ2
    case 0: f();             ...
        break;             if (c < 0) goto done
    case 1: g();             if (c > 2) goto done
        break;             targ = jtarr[c]
    case 2: h();             goto targ /* this is not C */
        break;             targ0: f ()
                                goto done
                                targ1: g ()
                                    goto done
                                targ2: h ()
                                    done: ...
}
```

Compiling Arrays

Array declaration

- Store name, size, and base type in symbol table

Array allocation

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

Array referencing

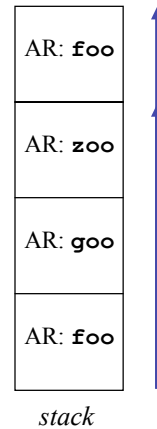
– e.g., **A[i]** → ***(&A + i * sizeof(A_elem))**

```
↓
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* (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)

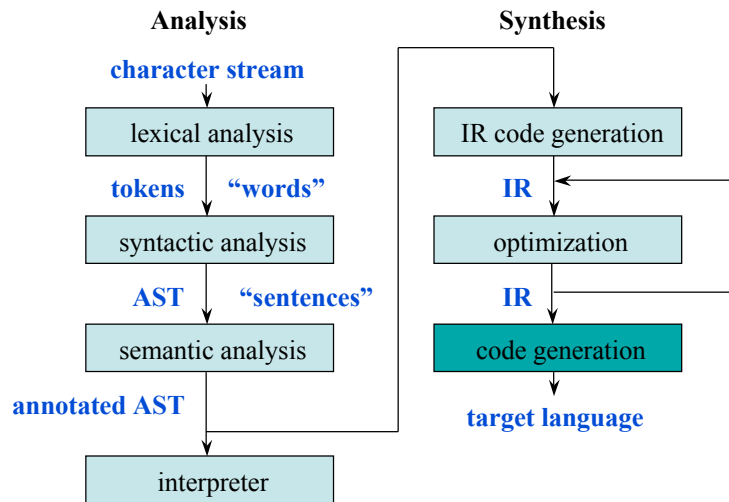
Code generation for procedures

- 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

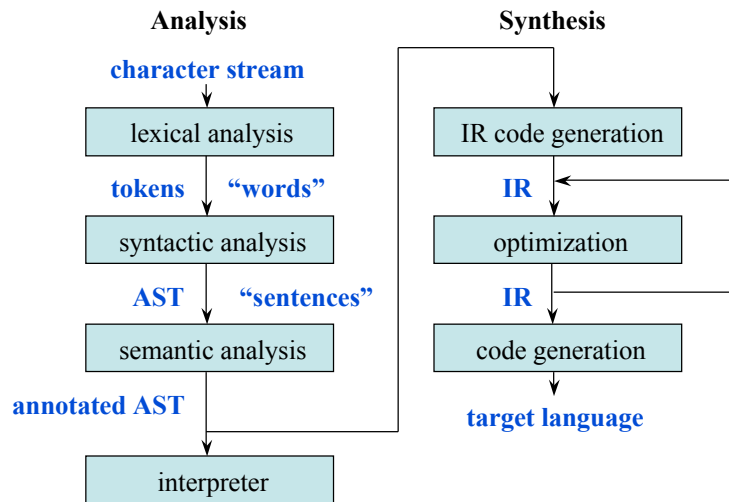
Conceptually easy

- 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. . .

Structure of a Typical Compiler



Question

Q: How can I have fun without the use of alcohol?

A: gcc -S

Example

```
% gcc -O0 -S file.c

int main() {
    int var1, var2, var3;

    var1 = 123;
    var2 = 3;
    var3 = var1/var2 + 1;

    return var3;
}
```

Tada! (file.s)

```
_main:
    stmw r30,-8(r1)
    stwu r1,-64(r1)
    mr r30,r1
    li r0,123
    stw r0,32(r30)
    li r0,3
    stw r0,28(r30)
    lwz r2,32(r30)
    lwz r0,28(r30)
    divw r2,r2,r0
    addi r0,r2,1
    stw r0,24(r30)
    lwz r0,24(r30)
    mr r3,r0
    lwz r1,0(r1)
    lmw r30,-8(r1)
    blr
```

Concepts

Compilation stages

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

Representations

- AST, low-level IR (RTL)

Coming Attractions

Optimizing compilers reason about program behavior

Q: With a low level intermediate representation (IR), how does a compiler reason about the behavior of a program?
– e.g. What is the structure of a program?
– e.g. What paths are possible in a program?

A: Control-flow analysis (next lecture)

Q: How do humans understand a program's behavior?

A: We typically simulate its behavior.

Q: How do compilers “understand” a program's behavior?

A: Data-flow analysis (lecture 4)

Next Time

Lecture

- Control flow analysis

Next Tuesday

- Discussion of Sites paper
- Discussion questions online
- Email me answers