Compiler Design

Lecture 22: Summary

Christophe Dubach
Winter 2021
Summary

Front-end

Parsing

Program Representation and Semantic

Backend

Code Generation

Register Allocator

Instruction Selection

Object Oriented Support & Garbage Collection

Object Oriented Support

Garbage Collection

The road ahead
Summary

Front-end

Parsing

Program Representation and Semantic

Backend

Code Generation

Register Allocator

Instruction Selection

Object Oriented Support & Garbage Collection

Object Oriented Support

Garbage Collection

The road ahead
The Lexer

- Maps character stream into words/tokens
- Typical tokens: number, identifier, +, -, new, while, if, ...
- Assign a syntactic category to each token:
  - $x = x + y$; becomes ID(x) EQ ID(x) PLUS ID(y) SC

Typically, tokens are described using regular expressions.

Example

| IDENTIFIER ::= ( 'a' | ... | 'Z' | '_' ) ( '0' | ... | '9' | 'a' | ... | 'Z' | '_' ) * |
The Regular Expression corresponds to a finite state automata:

\[ '0'|...'9'|a'|...'|Z'|_\]

It is possible to systematically generate a lexer for any regular expression. This can be done in three steps:

1. regular expression (RE) \(\rightarrow\) non-deterministic finite automata (NFA)
2. NFA \(\rightarrow\) deterministic finite automata (DFA)
3. DFA \(\rightarrow\) generated lexer
Table encoding

|      | 'a'|...|'Z'| | ` ' |...|'9' | | other |
|------|---|---|---| | ---|---|---|---|
| $s_0$ | $s_1$ | error | error |
| $s_1$ | $s_1$ | $s_1$ | end |

Skeleton recogniser

c = next character
state = $s_0$
while (c ≠ EOF)
    state = $\delta$(state, c)
    c = next character
if (state final)
    return success
else
    return error
The Parser

- Checks grammatical correctness of the stream of words/tokens produced by the lexer
- Outputs the AST (Abstract Syntax Tree) which represents the input program
Writing a recursive descent parser:

1. Express the language syntax as an LL(k) CFG;
2. Encode precedence’
3. Left factorize the grammar if necessary;
4. Remove left recursion from the grammar if present;
5. Write the recursive parser using at most $k$ lookaheads.

LL(k) grammar

- Left-to-Right parsing;
- Leftmost derivation; (i.e. apply production for leftmost non-terminal first)
- Only $k$ symbols of lookahead to make decision.

Your parser will never have to backtrack!
$\Rightarrow O(N)$ time complexity
## Top-Down vs. Bottom-Up Parsers

<table>
<thead>
<tr>
<th>Top-Down Parser</th>
<th>Bottom-Up Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Top-Down parser builds a derivation by working from the start symbol to the input sentence.</td>
<td>A Bottom-Up parser builds a derivation by working from the input sentence back to the start symbol.</td>
</tr>
</tbody>
</table>
**Leftmost vs Rightmost derivation**

**Example: CFG**

<table>
<thead>
<tr>
<th>Production</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal ::= a A B e</td>
<td></td>
</tr>
<tr>
<td>A ::= A b c</td>
<td>b</td>
</tr>
<tr>
<td>B ::= d</td>
<td></td>
</tr>
</tbody>
</table>

**Leftmost derivation**

- Goal
- aABe
- aAbcBe
- abbcBe
- abbcde

**Rightmost derivation**

- Goal
- aABe
- aAde
- aAbcde
- abbcde

**LL parsers**

**LR parsers**
LL grammars subset of LK grammar since they cannot be left-recursive!
Summary

Front-end

Parsing

Program Representation and Semantic

Backend

Code Generation

Register Allocator

Instruction Selection

Object Oriented Support & Garbage Collection

Object Oriented Support

Garbage Collection

The road ahead
Concrete Syntax Tree (Parse Tree)

CFG for arithmetic expressions

\[
\begin{align*}
\text{Expr} &::= \text{Term} \ (('+' | '-' ) \text{Expr} | \epsilon) \\
\text{Term} &::= \text{Factor} \ (('\times' | '/') \text{Term} | \epsilon) \\
\text{Factor} &::= \text{number} | \text{id} | '(' \text{Expr} ')'
\end{align*}
\]

Concrete Syntax Tree for 5 * 3

The concrete syntax tree contains a lot of unnecessary information.
Example: abstract grammar for arithmetic expressions

```
Expr ::= BinOp | intLiteral | id
BinOp ::= Expr Op Expr
Op ::= add | sub | mul | div

5 * 3
```

This is the Abstract Syntax Tree
When pattern matching not available in the host programming language, using visitor design pattern to implement double-dispatch mechanism.

Visitor Interface

```java
interface Visitor<T> {
    T visitIntLiteral(IntLiteral il);
    T visitBinOp(BinOp bo);
}
```

AST classes

```java
abstract class Expr {
    abstract <T> T accept(Visitor<T> v);
}

class IntLiteral extends Expr {
    ...
    <T> T accept(Visitor<T> v) {
        return v.visitIntLiteral(this);
    }
}
class BinOp extends Expr {
    ...
    <T> T accept(Visitor<T> v) {
        return v.visitBinOp(this);
    }
}
```
Link variable uses and function calls to their declaration.

```c
int i;
int j;
void foo() { ... };
void main(int i) {
    int j;
    i;
    {
        int j;
        j;
    }
    j;
    foo();
}
```

Use symbol table and scope information to track identifiers.
Different languages, different approaches

<table>
<thead>
<tr>
<th></th>
<th>strong</th>
<th>weak</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>Java</td>
<td>C/C++</td>
</tr>
<tr>
<td>dynamic</td>
<td>Python</td>
<td>JavaScript</td>
</tr>
</tbody>
</table>

Use typing rules to formalize. For instance:

\[
\text{IntLit} \quad \vdash i : \text{int}
\]

\[
\text{BinOp} \quad \vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \quad \vdash e_1 + e_2 : \text{int}
\]

Traverse the AST depth-first, post-order, to assign a type to every expression.
Summary

Front-end
  Parsing
  Program Representation and Semantic

Backend
  Code Generation
  Register Allocator
  Instruction Selection

Object Oriented Support & Garbage Collection
  Object Oriented Support
  Garbage Collection

The road ahead
Summary

Front-end
- Parsing
- Program Representation and Semantic

Backend
- Code Generation
- Register Allocator
- Instruction Selection

Object Oriented Support & Garbage Collection
- Object Oriented Support
- Garbage Collection

The road ahead
The Back end

- Translate IR into target machine code (or something resembling it);
- Choose instructions to implement each IR operation;
- Decide which value to keep in registers.
To simplify, use virtual registers and convert to arch. registers later.

Simple AST traversal:

```java
Register visitBinOp(BinOp bo) {
    Register lhsReg = bo.lhs.accept(this);
    Register rhsReg = bo.rhs.accept(this);
    Register resReg = new VirtualRegister();
    switch (bo.op) {
        case ADD:
            emit("add", resReg, lhsReg, rhsReg);
            break;
        case MUL:
            emit("mult", lhsReg, rhsReg);
            emit("mflo", resReg);
            break;
        ...
    }
    return resReg;
}
```
Control-Flow structures

- loops;
- logical operators (short-circuit evaluation);
- if-then-else.

Use simple code pattern / shapes and emits labels:
Memory allocation

A compiler keeps track of where variables are allocated, and allocate each of them.

```c
char c; static
int arr [4]; static
void foo () {
    int arr2 [3]; stack
    int* ptr = (int*) malloc (sizeof(int)*2); heap
    ...
    {
        int b; stack/register
        ...
        bar ("hello"); static
    }
    ...
}
```
Dealing with function calls

Things can get quite complex when dealing with function call:

- The compiler must manage the $sp and $fp correctly;
- Save the registers used by the function on the stack;
- As well as the stack pointer and frame pointer.
- Some arguments must be copied while others are passed by reference.

All this of course depends on the language and the target machine!
Summary

Front-end
- Parsing
- Program Representation and Semantic

Backend
- Code Generation
- Register Allocator
- Instruction Selection

Object Oriented Support & Garbage Collection
- Object Oriented Support
- Garbage Collection

The road ahead
Register allocation in a nutshell

1. Pseudo-assembly:

```
    a = 0
    L1: b = a + 1
        c = c + b
        a = b * 2
    if (a < 9) goto L1
    return c
```

2. Control flow graph:

![Control Flow Graph](image)

3. Liveness:

<table>
<thead>
<tr>
<th>node</th>
<th>out</th>
<th>in</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ac</td>
<td>ac</td>
</tr>
<tr>
<td>4</td>
<td>ac</td>
<td>bc</td>
</tr>
<tr>
<td>3</td>
<td>bc</td>
<td>bc</td>
</tr>
<tr>
<td>2</td>
<td>bc</td>
<td>ac</td>
</tr>
<tr>
<td>1</td>
<td>ac</td>
<td>c</td>
</tr>
</tbody>
</table>

4. Interference graph:

![Interference Graph](image)
5. Graph colouring:

![Graph Colouring Diagram]

6. Virtual to architectural registers

Possible mapping:
- $a \rightarrow \$t0$
- $b \rightarrow \$t0$
- $c \rightarrow \$t1$

(pseudo-)assembly final code:

```
\$t0 = 0
L1: \$t0 = \$t0 + 1
\$t1 = \$t1 + \$t0
\$t0 = \$t0 \times 2
if (\$t0 < 9) goto L1
return \$t1
```

Use Chaitin algorithm for graph colouring.
Register allocation challenges

- Graph colouring can be slow:
  - Production compilers prefer other techniques such as linear scan.
- When it is not possible to colour the graph, need to spill:
  - this is what is crucial for producing good quality code;
  - a bit of a dark art.
Summary

Front-end
  Parsing
  Program Representation and Semantic

Backend
  Code Generation
  Register Allocator
  Instruction Selection

Object Oriented Support & Garbage Collection
  Object Oriented Support
  Garbage Collection

The road ahead
Goal: produce “smarter” code by exploiting instruction set.

**Peephole matcher**

Steps of the Simplifier  
(3-operation window)

**LLIR Code**

\[
\begin{align*}
    r_{10} &\leftarrow 2 \\
    r_{11} &\leftarrow @y \\
    r_{12} &\leftarrow r_0 + r_{11} \\
    r_{13} &\leftarrow \text{MEM}(r_{12}) \\
    r_{14} &\leftarrow r_{10} \times r_{13} \\
    r_{15} &\leftarrow @x \\
    r_{16} &\leftarrow r_0 + r_{15} \\
    r_{17} &\leftarrow \text{MEM}(r_{16}) \\
    r_{18} &\leftarrow r_{17} - r_{14} \\
    r_{19} &\leftarrow @w \\
    r_{20} &\leftarrow r_0 + r_{19} \\
    \text{MEM}(r_{20}) &\leftarrow r_{18}
\end{align*}
\]

\[
\begin{align*}
    r_{10} &\leftarrow 2 \\
    r_{12} &\leftarrow r_0 + @y \\
    r_{13} &\leftarrow \text{MEM}(r_{12}) \\
    r_{14} &\leftarrow r_{10} \times r_{13} \\
    r_{13} &\leftarrow \text{MEM}(r_0 + @y) \\
    r_{14} &\leftarrow r_{10} \times r_{13}
\end{align*}
\]
Tree Pattern-Matcher

Sequences with Cost of 2

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6: Reg → LAB&lt;sub&gt;1&lt;/sub&gt;</td>
<td>loadI @G ⇒ r&lt;sub&gt;1&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>11: Reg → REF(+(Reg&lt;sub&gt;1&lt;/sub&gt;,NUM&lt;sub&gt;2&lt;/sub&gt;))</td>
<td>loadAI r&lt;sub&gt;1&lt;/sub&gt;,12 ⇒ r&lt;sub&gt;j&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>8: Reg → NUM&lt;sub&gt;1&lt;/sub&gt;</td>
<td>loadI 12 ⇒ r&lt;sub&gt;1&lt;/sub&gt;</td>
<td></td>
</tr>
<tr>
<td>14: Reg → REF(+(LAB&lt;sub&gt;1&lt;/sub&gt;,Reg&lt;sub&gt;2&lt;/sub&gt;))</td>
<td>loadAI r&lt;sub&gt;1&lt;/sub&gt;,@G ⇒ r&lt;sub&gt;j&lt;/sub&gt;</td>
<td></td>
</tr>
</tbody>
</table>
Summary

Front-end
  Parsing
  Program Representation and Semantic

Backend
  Code Generation
  Register Allocator
  Instruction Selection

Object Oriented Support & Garbage Collection
  Object Oriented Support
  Garbage Collection

The road ahead
Summary

Front-end
  Parsing
  Program Representation and Semantic

Backend
  Code Generation
  Register Allocator
  Instruction Selection

Object Oriented Support & Garbage Collection

Object Oriented Support

Garbage Collection

The road ahead
Single inheritance

Easy to support:

- Layout fields from superclass sequentially;
- Use a single virtual table.

```c
class A {
    void foo() { printf("foo_a") }
    void bar() { printf("bar_a") }
}

class B extends A {
    void bar() { printf("bar_b") }
    void baz() { printf("baz_b") }
}
```
Multiple inheritance

Much more complex to handle. Need to embed the superclass object layout and use multiple virtual tables.
Summary

Front-end
- Parsing
- Program Representation and Semantic

Backend
- Code Generation
- Register Allocator
- Instruction Selection

Object Oriented Support & Garbage Collection
- Object Oriented Support
- Garbage Collection

The road ahead
Goal: find objects/records that can never be reached.

Three main approaches:
- Reference counting;
- Mark and sweep;
- Stop and copy;

Stop-and-Copy, with a generational collection is pretty good.
Summary

Front-end
- Parsing
- Program Representation and Semantic

Backend
- Code Generation
- Register Allocator
- Instruction Selection

Object Oriented Support & Garbage Collection
- Object Oriented Support
- Garbage Collection

The road ahead
In this course, we have only scratched the surface of the world of compilers. Compilation is still a very active research field and there is plenty of development.

If you want to gain experience with industry compilers:

- For C like languages: LLVM
- For Java like languages: GraalVM / Truffle (from Oracle Labs)
- For JavaScript: V8

Hot compiler IRs:

- MLIR (connected to LLVM)
- WebAssembly (virtual assembly for the web)
Courses you may also like:

- COMP 764: “High-Level Hardware Synthesis” (TBD, Winter 2022)
- ECSE 427 / COMP 310: Operating Systems
- COMP 409: Concurrent Programming

What to read next:

The “Dragon book”:
Compilers: Principles, Techniques, and Tools
Alfred Aho*, Monica Lam, Ravi Sethi, Jeffrey Ullman*
*ACM Turing Award Winners, 2021
Research in my group (COMP 400, ECSE 498, SURE/SURA)

- Parallel programming abstractions based on Lambda calculus
- Rewrite-based optimizations
- High performance code generation
- High-level hardware synthesis

Looking for job/internship related to compilation?

- https://github.com/mgaudet/CompilerJobs
The end