CMPT 379
Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop
Building a compiler

- Programming languages have a lot in common
- Do not write a compiler for each language
- Create a general mathematical model for all languages: implement this model
- Each language compiler is built using this general model (so-called compiler compilers)
  - yacc = yet another compiler compiler
- Code optimization ideas can also be shared across languages
Building a compiler

- The cost of compiling and executing should be managed
- No program that violates the definition of the language should escape
- No program that is valid should be rejected
Building a compiler

• Requirements for building a compiler:
  – Symbol-table management
  – Error detection and reporting

• Stages of a compiler:
  – Analysis (front-end)
  – Synthesis (back-end)
Stages of a Compiler

• Analysis (Front-end)
  – Lexical analysis
  – Syntax analysis (parsing)
  – Semantic analysis (type-checking)

• Synthesis (Back-end)
  – Intermediate code generation
  – Code optimization
  – Code generation
Lexical Analysis

• Also called scanning, take input program string and convert into tokens

• Example:

double f = sqrt(-1);

<table>
<thead>
<tr>
<th>Token Type</th>
<th>Token Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_DOUBLE</td>
<td>(&quot;double&quot;)</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>(&quot;f&quot;)</td>
</tr>
<tr>
<td>T_OP</td>
<td>(&quot;=&quot;)</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>(&quot;sqrt&quot;)</td>
</tr>
<tr>
<td>T_LPAREN</td>
<td>(&quot;(&quot;)</td>
</tr>
<tr>
<td>T_OP</td>
<td>(&quot;-&quot;)</td>
</tr>
<tr>
<td>T_INTCONSTANT</td>
<td>(&quot;1&quot;)</td>
</tr>
<tr>
<td>T_RPAREN</td>
<td>(&quot;)&quot;)</td>
</tr>
<tr>
<td>T_SEP</td>
<td>(&quot;;&quot;)</td>
</tr>
</tbody>
</table>
Syntax Analysis

• Also called *parsing*
• Describe the set of strings that are programs using a grammar
• Pick the simplest grammar formalism possible (but not too simple)
  – Finite-state machines (Regular grammars)
  – Deterministic Context-free grammars
  – Context-free grammars
• Structural validation
• Creates parse tree or derivation
Derivation of \text{sqrt}(-1)

Expression -> UnaryExpression
Expression -> FuncCall
Expression -> T\_INTCONSTANT
UnaryExpression -> T\_OP Expression
FuncCall -> T\_IDENT T\_LPAREN Expression T\_RPAREN

Expression
  -> FuncCall
  -> T\_IDENT T\_LPAREN Expression T\_RPAREN
  -> T\_IDENT T\_LPAREN UnaryExpression T\_RPAREN
  -> T\_IDENT T\_LPAREN T\_OP Expression T\_RPAREN
  -> T\_IDENT T\_LPAREN T\_OP T\_INTCONSTANT T\_RPAREN
Parse Trees

```
Expression
  FuncCall
    T_IDENT  T_LPAREN  Expression  T_RPAREN
    sqrt ( )

UnaryExpression
  T_OP  Expression
    -
    T_INTCONSTANT 1
```
Abstract Syntax Tree

```
[Expr(
    value=Call(
        func=Attribute(
            value=Name(
                id='math',
                ctx=Load()
            ),
            attr='sqrt',
            ctx=Load()
        ),
        args=[Num(n=-1)],
        keywords=[],
        starargs=None,
        kwargs=None
    )
)
]```
Semantic analysis

• “does it make sense”? Checking semantic rules,
  – Is there a main function?
  – Is variable declared?
  – Are operand types compatible? (coercion)
  – Do function arguments match function declarations?

• Type checking: operational or denotational semantics

• Static vs. run-time semantic checks
  – Array bounds, return values do not match definition
Intermediate Code Generation

• Three-address code (TAC)

\[
j = 2 \times i + 1; \\
\text{if } (j \geq n) \\
\quad j = 2 \times i + 3; \\
\text{return } a[j];
\]

\[
_t1 = 2 \times i \\
_t2 = _t1 + 1 \\
j = _t2 \\
_t3 = j < n \\
\text{if } _t3 \text{ goto L0} \\
_t4 = 2 \times i \\
_t5 = _t4 + 3 \\
j = _t5 \\
\text{L0: } _t6 = a[j] \\
\text{return } _t6
\]
Code Optimization

• Example

\[ \_t1 = 2 \times i \]
\[ \_t2 = \_t1 + 1 \]
\[ j = \_t2 \]
\[ \_t3 = j < n \]
if \_t3 goto L0
\[ \_t4 = 2 \times i \]
\[ \_t5 = \_t4 + 3 \]
\[ j = \_t5 \]
L0: \[ \_t6 = a[j] \]
return \_t6

\[ \_t1 = 2 \times i \]
\[ \_t2 = \_t1 + 1 \]
\[ j = \_t2 \]
\[ \_t3 = j < n \]
if \_t3 goto L0
\[ \_t4 = 2 \times i \]
\[ \_t5 = \_t4 + 3 \]
\[ j = \_t5 \]
L0: \[ \_t6 = a[j] \]
return \_t6
Object code generation

• Example: \( a \) in \$a0, \( i \) in \$a1, \( n \) in \$a2

\[
_t1 = 2 \times i \\
j = _t1 + 1 \\
_t3 = j < n \\
if \_t3 \text{ goto L0} \\
j = _t1 + 3
\]

\[
mulo \_t1, \_a0, 2 \\
add \_s0, \_t1, 1 \\
seq \_t2, \_s0, \_a2 \\
beq \_t2, 1, L0 \\
add \_s0, \_t1, 3
\]
Bootstrapping a Compiler

• Machine code at the beginning
• Make a simple subset of the language, write a compiler for it, and then use that subset for the rest of the language definition
• Bootstrap from a simpler language
  – C++ ("C with classes")
• Interpreters
• Cross compilation
Modern challenges

• Instruction Parallelism
  – Out of order execution; branch prediction

• Parallel algorithms:
  – Grid computing,
  – multi-core computers

• Memory hierarchy: register, cache, memory

• Binary translation, e.g. x86 to VLIW

• New computer architectures, e.g. streaming algorithms

• Hardware synthesis / Compiled simulations
Wrap Up

• Analysis/Synthesis
  – Translation from string to executable

• Divide and conquer
  – Build one component at a time
  – Theoretical analysis will ensure we keep things simple and correct
  – Create a complex piece of software