Lecture 3. Implementing a Compiler

Wei Le

2015.8
Phases and Passes

- Passes: whether the compilers visit the same code (e.g. statement) more than one time

- Phases - conceptual stages, Symbol table coordinating information between phases, e.g., lexical analysis, syntactical analysis
Phases and Passes

- Passes: whether the compilers visit the same code (e.g. statement) more than one time
- Single pass compiler:
  - Limited local optimization
  - Simplicity and efficiency
  - Pascal
  - All phases of compiler interleaved
  - Compilation driven by parser
  - Scanner acts as subroutine of parser – ”give me next token”
  - As each phrase recognised by parser, it calls semantic routines to process declarations, check for semantic errors and generate code

- Phases - conceptual stages, Symbol table coordinating information between phases, e.g., lexical analysis, syntactical analysis
Phases and Passes

- Passes: whether the compilers visit the same code (e.g. statement) more than one time

- Single pass compiler:
  - Limited local optimization
  - Simplicity and efficiency
  - Pascal
  - All phases of compiler interleaved
  - Compilation driven by parser
  - Scanner acts as subroutine of parser – ”give me next token”
  - As each phrase recognised by parser, it calls semantic routines to process declarations, check for semantic errors and generate code

- Multi-pass compiler:
  - Modern compiler requires multiple pass
  - Multi-pass allows complete separation of phases, more modular, easier to develop, more portable
  - Main forms of IR: building AST, IR implies multiple pass
  - pass one: collect information, pass two – code transformation
  - example points-to analysis and constant propagation

- Phases - conceptual stages, Symbol table coordinating information between phases, e.g., lexical analysis, syntactical analysis
1. $X = 5$
2. $*p = \ldots$  // $p$ may or may not point to $X$
3. $\ldots = X$

*Constant propagation*: assume $p$ does point to $X$ (i.e., in statement 3, $X$ cannot be replaced by 5).

*Dead Code Elimination*: assume $p$ does not point to $X$ (i.e., statement 1 cannot be deleted).
Current Compiler Research

Conferences:
- (PLDI) Programming Language Design and Implementation.
- (CC) Compiler construction
- (CGO) Code Generation and Optimization

Topics:
- Finding bugs in compilers
- Automatic parallelization
- Approximate computation
- Loop analysis
- Machine learning for code optimizations
- Call graph construction
Programming Languages

- Why so many programming languages?

- Requirements: expressiveness, controlling the resources
Programming Languages

- Why so many programming languages?
- Applications, people think differently
  - Scientific computation: float pointers, arrays and operations, parallelism (Fortune–formula translations, Julia)
  - Business applications: security, persistence, (SQL)
  - System programming: manage resources, real-time constraints (reasoning about time) (C)
- Requirements: expressiveness, controlling the resources
Why new programming languages?
Why new programming languages?

Cost
- design the language
- implement the language: compiler
- train the developers to use the language (hours, text books)

Prediction
- wildly used languages are slow to change
- easy to start a new language
- if the productivity exceed the time of training (easy to use, largely improve the productivity – adopted to fill a void, e.g., Mobile computing, Internet languages)
- new languages tend to look like an old language (economic reasons, Java v.s C++)
Language Design

What makes a good programming language? (research question?)

▶ No universally accepted metrics for design
▶ A good language is the one people use? NO. Visual Basic and COBOL are the best languages?
▶ Good language design is hard
<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Readability</td>
</tr>
<tr>
<td>Simplicity</td>
<td>*</td>
</tr>
<tr>
<td>Data types</td>
<td>*</td>
</tr>
<tr>
<td>Syntax design</td>
<td>*</td>
</tr>
<tr>
<td>Abstraction</td>
<td></td>
</tr>
<tr>
<td>Expressivity</td>
<td>*</td>
</tr>
<tr>
<td>Type checking</td>
<td></td>
</tr>
<tr>
<td>Exception handling</td>
<td></td>
</tr>
</tbody>
</table>
Compilers

- More needed and more complex
- Driven by increasing gap between new languages and new architectures
- Venerable and healthy area
Implementing a Compiler

- Algorithms written and tuned by developers: efficient, clean code, flexibility
- Tool assisted compiler building: easy to build
- GCC/Clang: hand-written recursive descendant parser (gcc was once used Bison), 1.5% speedup, further reading https://gcc.gnu.org/wiki/New_C_Parser
Compiler Tools

Compiler-compiler or compiler generator is a programming tool that creates a parser, interpreter, or compiler from some form of formal description of a language and machine:

Scanner generator: Lex, JLex, Flex
- Input: a declarative specification (tokens, regular expressions)
- Output: lexical analyzers

Parser generator: YACC, Bison
- Input: BNF grammars – specification of syntactic structures
- Output: Parsing programs (parser)
Cool: Classroom Object Oriented Language

► Designed to be implementable in a short time
► Give a taste of implementation of modern languages
  ► Abstraction
  ► Static typing
  ► Reuse (inheritance)
  ► Memory management ...
► But many things are left out
Abstraction

- Abstraction: detached from concrete details
- Abstraction necessary to build software systems
  - languages and compilers: Higher-level code, few machine dependencies
  - subroutines: Abstract interface to behavior
  - modules: Export interfaces; hide implementation
  - abstract data types: Bundle data with its operations
Types

- Originally, few types
  - FORTRAN: scalars, arrays
  - LISP: every thing is dynamically typed
- Realization: Types help
  - Allow the programmer to express abstraction
  - Allow the compiler to check against many frequent errors
  - Sometimes to the point that programs are guaranteed: "safe"
- More recently
  - Lots of interest in types
  - Experiments with various forms of parameterization
  - Best developed in functional programming
Reuse

- **Reuse** = exploit common patterns in software systems
- **Goal**: mass-produced software components
- **Two popular approaches**
  - Type parameterization: \((\text{List}(\text{int}), \text{List}(\text{double}))\)
  - Classes and inheritance: C++ derived classes
- **Inheritance allows**:
  - Specialization of existing abstraction
  - Extension, modification, hiding behavior
A Simple Example

class Point {
    x : Int ← 0;
    y : Int ← 0;
};

- Cool programs are sets of class definitions
  - A special class Main with a special method main
  - No separate notion of subroutine

- class = a collection of attributes and methods
- Instances of a class are objects
Cool Objects

class Point {
    x : Int ← 0;
    y : Int; /* use default value */
}

• The expression "new Point" creates a new object of class Point

• An object can be thought of as a record with a slot for each attribute

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Methods

- A class can also define methods for manipulating the attributes

```java
class Point {
    x : Int ← 0;
    y : Int ← 0;
    movePoint(newx : Int, newy : Int): Point {
        { x ← newx;
          y ← newy;
          self;
        } -- close block expression
    }; -- close method
}; -- close class
```

- Methods can refer to the current object using `self`
Information Hiding in Cool

- Methods are global
- Attributes are local to a class
  - They can only be accessed by the class’s methods
- Example:

  class Point {
    . . .
    getx () : Int { x };
    setx (newx : Int) : Int { x ← newx };
  };

Methods

- Each object knows how to access the code of a method
- As if the object contains a slot pointing to the code

```
  x  y  movePoint
  0  0  *
```

- In reality, implementations save space by sharing these pointers among instances of the same class

```
  x  y  methods
  0  0

  movePoint
  *
```
Inheritance

- We can extend points to colored points using subclassing => class hierarchy

```java
class ColorPoint inherits Point {
    color : Int ← 0;
    movePoint(newx : Int, newy : Int): Point {
        { color ← 0;
            x ← newx; y ← newy;
            self;
        }
    };
};
```

```
0 0 0 0 *
```
Cool Types

• Every class is a type
• Base classes:
  - `Int` for integers
  - `Bool` for boolean values: `true, false`
  - `String` for strings
  - `Object` root of the class hierarchy

• All variables must be declared
  - compiler infers types for expressions
Cool Type Checking

\[
x : A;
\]
\[
x \leftarrow \text{new} \ B;
\]

- Is well typed if \( A \) is an ancestor of \( B \) in the class hierarchy
  - Anywhere an \( A \) is expected a \( B \) can be used

- Type safety:
  - A well-typed program cannot result in runtime type errors
Method Invocation and Inheritance

- Methods are invoked by dispatch

- Understanding dispatch in the presence of inheritance is a subtle aspect of OO languages
  
  ```
  p : Point;
  p ← new ColorPoint;
  p.movePoint(1,2);
  ```

  - p has static type Point
  - p has dynamic type ColorPoint
  - p.movePoint must invoke the ColorPoint version
Method Invocation

- Example: invoke one-argument method m

1. Eval. e
2. Find class of e
3. Find code of m
4. Eval. argum. e'
5. Bind self and x
6. Run method
Other Expressions

• Expression language
  - every expression has a type and a value
  - Loops: while E loop E pool
  - Conditionals if E then E else E fi
  - Case statement case E of x : Type ⇒ E; ... esac
  - Arithmetic, logical operations
  - Assignment x ← E
  - Primitive I/O out_string(s), in_string(), ...

• Missing features:
  - arrays, floating point operations, exceptions, ...
Cool Memory Management

- Memory is allocated every time `new` is invoked

- Memory is deallocated automatically when an object is not reachable anymore
  - Done by the garbage collector (GC)
  - There is a Cool GC
Marple Scanner and Parser