COMP 520 - Compilers

Lecture 6 (Tue Jan 29)

Compiler Design and Implementation

• Please turn in at front of the class
  – Written assignment WA2

• Reading assignment for Thursday
  – Skim secn 4.4 pp 109 - 118

• PA1 due next Monday, Feb 4 by 11:59PM
  – Submission instructions later this week
Overview

• Today’s topics
  – Finish scanner discussion
    • Examine Triangle scanner
    • Automatic scanner generation
  – How compilers are organized
    • Steps in the compilation process
    • Illustration of the steps for a Triangle program
    • parts of the compiler and their relation to the compiler specification
    • a bird’s eye view of implementation
## Phases of compilation

<table>
<thead>
<tr>
<th>Phase</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lexical analysis</strong></td>
<td>character stream</td>
<td>tokens</td>
</tr>
<tr>
<td>(scanner)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>syntactic analysis</strong></td>
<td>tokens</td>
<td>abstract syntax tree (AST)</td>
</tr>
<tr>
<td>(parser)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Contextual analysis</strong></td>
<td>AST</td>
<td>decorated AST</td>
</tr>
<tr>
<td>(type checker / identifier resolution)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>optional</strong></td>
<td>decorated AST</td>
<td>decorated AST</td>
</tr>
<tr>
<td>Optimization</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Code generation</strong></td>
<td>decorated AST</td>
<td>machine instructions</td>
</tr>
</tbody>
</table>

Unified in PLPJ text
FE: Lexical Analysis (Triangle)

- Recognize the meaningful units in the source code
  - input: character stream
    ```
    ! this is part of a Triangle program
    while b do
    begin
      n := 0;
      b := false
    end
    ```
  - output: token stream
    ```
    while b do begin
      n := 0;
      b := false
    end
    ```
FE: Syntactic Analysis (Triangle)

- Use Triangle grammar to parse token stream into (concrete) syntax tree

```
Program ::= Single-Command eot
Command ::= Single-Command | Command ; Single-Command
Single-Command ::= while Expression do Single-Command | V-name := Expression | begin Command end | ...
Expression ::= Primary-Expression | ...
Primary-Expression ::= intLit | V-name | ...
V-name ::= ident
```

![Syntax Tree Diagram]
FE: Construct Abstract Syntax Tree

- While parsing concrete syntax tree, construct the more useful abstract syntax tree.

**AST “grammar” for a part of mini-Triangle**
FE: Construct Abstract Syntax Tree (Triangle)

Concrete Syntax Tree

Abstract Syntax Tree
FE: Contextual analysis

- Traverse AST
  - determine the declaration associated with each identifier referenced
  - determine the type of all expressions
  - record in AST

```
let
  var n: Integer;
  var c: Char
in
begin
  c := '&';
  n := n + 1
end
```

Decorated AST for sample program
BE: Optimization

- Restructure AST so that it corresponds to an equivalent but more efficient program
  - simple example: constant folding
    \[ x := y \times 0 \Rightarrow x := 0 \]
  - introduce temporaries to hold previously computed values

\[
\begin{align*}
f[i] & := f[i] + (m[i] \times m[j]) / \text{pow}(x[i] - x[j], 2); \\
f[j] & := f[j] - (m[i] \times m[j]) / \text{pow}(x[i] - x[j], 2);
\end{align*}
\]
BE: Code Generation

- Produce machine code
  - for abstract machine or physical machine

- Issues
  - location of program variables (stack, heap)
  - instruction selection and register allocation
  - linkage conventions (ABI: Application Binary Interface)

```
let
  var n: Integer
  var c: Char
in
  begin
    c := '&';
    n := n + 1
  end
```

Triangle

<table>
<thead>
<tr>
<th>PUSH 2</th>
<th>LOADL 38</th>
<th>STORE 1[SB]</th>
<th>LOAD 0[SB]</th>
<th>LOADL 1</th>
<th>CALL add</th>
<th>STORE 0[SB]</th>
<th>POP 2</th>
<th>HALT</th>
</tr>
</thead>
</table>
Implementation of multiple phases

- **Single-pass compiler**
  - economical in time and space
  - limited ability to implement language features and optimizations

- **Multi-pass compiler**
  - conceptually simpler and more versatile
  - requires more space and time

- **Wirth’s design principle**
  - design programming languages so that their compilers are simple
    - Pascal
    - C + “.h” header files

- **Programmers design principle**
  - but not too simple!
Traditional Two-Pass Compiler

- **Front end** creates AST
  - time complexity $O(n)$ or $O(n \log n)$ where $n$ is size of the program in characters

- **Back end** translates AST to target machine
  - $O(n)$ or $O(n \log n)$ time complexity for simple code generation
  - but most back end optimization problems are NP-hard
**UNCOL: the universal AST?**

- Suppose we have
  - \( n \) programming languages
  - \( m \) target machines
  - we need to construct \( n \times m \) compilers

- **A universal AST would allow us to**
  - construct \( n \) front-ends
  - construct \( m \) back ends
  - \( n + m \) components: much less work!

- **The Universal Computer Oriented Language: an elusive goal since 1960**
  - variation (evolution) in programming languages
  - variation (evolution) in hardware architecture
  - but hope springs eternal: JVM? .NET?