COMP 520 - Compilers

Lecture 6 (Tue Jan 31, 2017)

Compiler Design and Implementation

- Please turn in at front of the class
  - Written assignment WA2

- Reading assignment for Thursday
  - Review secn 4.4 pp 109 - 118

- PA1 due next Monday, Feb 6 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> (scanner)</td>
<td>character stream</td>
<td>tokens</td>
</tr>
<tr>
<td><strong>syntactic analysis</strong> (parser)</td>
<td>tokens</td>
<td>abstract syntax tree (AST)</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 Optimization</strong></td>
<td>decorated AST</td>
<td>decorated AST</td>
</tr>
<tr>
<td><strong>Code generation</strong></td>
<td>decorated AST</td>
<td>machine instructions</td>
</tr>
</tbody>
</table>

Unified in PLPJ text

Front end

Back end
FE: Lexical Analysis (Triangle)

- Recognize the meaningful units in the source code
  - input: character stream
  
  ```plaintext
  ! this is part of a Triangle program
  while b do
    begin
      n := 0;
      b := false
    end
  ```

  - output: token stream

  ```plaintext
  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

[Diagram of syntax tree]

Program

single-Command

while b do begin n := 0 ; b := false end

eot
FE: Construct Abstract Syntax Tree

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

```
Program ::= Command

Command ::= V-name := Expression
          | Identifier ( Expression )
          | Command ; Command
          | if Expression then Command else Command
          | while Expression do Command
          | let Declaration in Command

Expression ::= Integer-Literal
          | V-name
          | Operator Expression
          | Expression Operator Expression

V-name ::= Identifier

Declaration ::= const Identifier := Expression
              | var Identifier : Type-denoter
              | Declaration ; Declaration

Type-denoter ::= Identifier
```

**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

Decorated AST for sample program
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>
</tr>
</thead>
<tbody>
<tr>
<td>STORE 1[ SB]</td>
<td>LOAD 0[ SB]</td>
</tr>
<tr>
<td>LOADL 1</td>
<td>CALL add</td>
</tr>
<tr>
<td>STORE 0[ SB]</td>
<td>POP 2</td>
</tr>
<tr>
<td>HALT</td>
<td></td>
</tr>
</tbody>
</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?