Abstract view

source code → compiler → machine code

errors

Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- need format for object (or assembly) code

Big step up from assembler – higher level notations
Traditional two pass compiler

source code → front end → \textit{il} → back end → machine code

\textbf{Implications:}

- intermediate language (\textit{il})
- front end maps legal code into \textit{il}
- back end maps \textit{il} onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes \Rightarrow better code

\textit{Front end is } \mathcal{O}(n) \textit{ or } \mathcal{O}(n \log n)

\textit{Back end is NP-Complete}
A fallacy

Can we build \( n \times m \) compilers with \( n + m \) components?

- must encode \textit{all} the knowledge in each front end
- must represent \textit{all} the features in one \textit{il}
- must handle \textit{all} the features in each back end

\textit{Limited success with low-level ils}
Front end

Responsibilities:

- recognize legal procedure
- report errors
- produce \( il \)
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated
Scanner

- maps characters into tokens – the basic unit of syntax
  \[ x = x + y; \]
  becomes
  \[ \langle \text{id}, x \rangle = \langle \text{id}, x \rangle + \langle \text{id}, y \rangle ; \]
- character string for a token is a lexeme
- typical tokens: number, id, +, -, *, /, do, end
- eliminates white space (tabs, blanks, comments)
- a key issue is speed
  \[ \Rightarrow \text{use specialized recognizer (lex)} \]
Specifying patterns

A scanner must recognize various parts of the language’s syntax.

Some parts are easy:

white space
    some combination of 
    and \texttt{tab}

keywords and operators
    specified as literal patterns — \texttt{do}, \texttt{end}

comments
    opening and closing delimiters — \texttt{/* \cdots */}
Specifying patterns

Other parts are much harder:

_identifiers_

alphabetic followed by \( k \) alphanumerics
\((-, \$, \&, \ldots)\)

_numbers_

integers — 0 or digit from 1-9 followed by digits from 0-9
decimals — integer “.” digits from 0-9
reals — (integer or decimal) “E” (+ or -) digits from 0-9
complex — “(" real "," real ")"

We need a powerful notation to specify these patterns.
### Definitions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>union of $L$ and $M$</strong></td>
<td>$L \cup M = { s \mid s \in L \text{ or } s \in M }$</td>
</tr>
<tr>
<td>written $L \cup M$</td>
<td></td>
</tr>
<tr>
<td><strong>concatenation</strong></td>
<td></td>
</tr>
<tr>
<td>of $L$ and $M$</td>
<td>$LM = { st \mid s \in L \text{ and } t \in M }$</td>
</tr>
<tr>
<td>written $LM$</td>
<td></td>
</tr>
<tr>
<td><strong>Kleene closure of $L$</strong></td>
<td>$L^* = \bigcup_{i=0}^{\infty} L^i$</td>
</tr>
<tr>
<td>written $L^*$</td>
<td></td>
</tr>
<tr>
<td><strong>positive closure of $L$</strong></td>
<td>$L^+ = \bigcup_{i=1}^{\infty} L^i$</td>
</tr>
<tr>
<td>written $L^+$</td>
<td></td>
</tr>
</tbody>
</table>

Aho, Sethi, and Ullman, Figure 3.8
Patterns are often specified as *regular languages*. Notations used to describe a regular language (or a regular set) include both *regular expressions* and *regular grammars*.

**Regular expressions (over an alphabet \( \Sigma \)):**

1. \( \epsilon \) is a \( \text{RE} \) denoting the set \( \{ \epsilon \} \)
2. if \( a \in \Sigma \), then \( a \) is a \( \text{RE} \) denoting \( \{a\} \)
3. if \( r \) and \( s \) are \( \text{RE}s \), denoting \( L(r) \) and \( L(s) \), then:
   
   \[
   (r) \text{ is a \( \text{RE} \) denoting } L(r) \\
   (r) | (s) \text{ is a \( \text{RE} \) denoting } L(r) \cup L(s) \\
   (r)(s) \text{ is a \( \text{RE} \) denoting } L(r)L(s) \\
   (r)^* \text{ is a \( \text{RE} \) denoting } L(r)^*
   \]

If we adopt a *precedence* for operators, the extra parentheses can go away. We assume *closure*, then *concatenation*, then *alternation* as the order of precedence.
**RE examples**

**identifier**

\[
\begin{align*}
letter & \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z) \\
\text{digit} & \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9) \\
id & \rightarrow letter (letter \mid digit)^*
\end{align*}
\]

**numbers**

\[
\begin{align*}
\text{integer} & \rightarrow \\
& (\pm \mid - \mid \epsilon) (0 \mid (1 \mid 2 \mid 3 \mid \ldots \mid 9) (\text{digit})^*) \\
\text{decimal} & \rightarrow \text{integer} \cdot (\text{digit})^* \\
\text{real} & \rightarrow (\text{integer} \mid \text{decimal}) \ E \ (\pm \mid -) (\text{digit})^+ \\
\text{complex} & \rightarrow "(" \text{real} \ "\)," \text{real} \ "")"
\end{align*}
\]

*Numbers can get much more complicated*

Most programming language tokens can be described with regular expressions.

We can use regular expressions to automatically build scanners.
Parser:

- recognize context-free syntax
- guide context-sensitive analysis
- construct *il*(s)
- produce meaningful error messages
- attempt error correction

*Parser generators mechanize much of the work*
Grammar

Context-free syntax is specified with a grammar.

<sheep noise> ::= baa
    | baa <sheep noise>

This grammar defines the set of noises that a sheep makes under normal circumstances.
The format is called Backus-Naur form. (BNF)

Formally, a grammar $G = (S, N, T, P)$

$S$ is the start symbol
$N$ is a set of non-terminal symbols
$T$ is a set of terminal symbols
$P$ is a set of productions or rewrite rules
$(P : N \rightarrow N \cup T)$
Substitution

Context free syntax can be put to better use.

1  <goal> ::= <expr>
2  <expr> ::= <expr> <op> <term>
3          | <term>
4  <term> ::= number
5          | id
6  <op>   ::= +
7          | -

This grammar defines simple expressions with addition and subtraction over the tokens id and number.

\[ S = <goal> \]
\[ T = \text{number}, \text{id}, +, - \]
\[ N = <goal>, <expr>, <term>, <op> \]
\[ P = 1, 2, 3, 4, 5, 6, 7 \]
Parse tree

Given a grammar, valid sentences can be derived by repeated substitution.

<table>
<thead>
<tr>
<th>Prod’n.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;goal&gt;</td>
</tr>
<tr>
<td>1</td>
<td>&lt;expr&gt;</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; &lt;op&gt; &lt;term&gt;</td>
</tr>
<tr>
<td>3</td>
<td>&lt;term&gt; + 2 - y</td>
</tr>
<tr>
<td>4</td>
<td>&lt;expr&gt; &lt;op&gt; 2 - y</td>
</tr>
<tr>
<td>5</td>
<td>x + 2 - y</td>
</tr>
<tr>
<td>6</td>
<td>&lt;expr&gt; + 2 - y</td>
</tr>
<tr>
<td>7</td>
<td>&lt;expr&gt; - y</td>
</tr>
<tr>
<td>2</td>
<td>&lt;expr&gt; &lt;op&gt; &lt;term&gt; - y</td>
</tr>
<tr>
<td>5</td>
<td>&lt;expr&gt; &lt;op&gt; y</td>
</tr>
</tbody>
</table>

To recognize a valid sentence in some cfg, we reverse this process and build up a parse.
Parse tree

A parse can be represented by a tree, called a parse tree or a syntax tree.

```
expr
   /\  \
  /   \  
term  +  term
```

Obviously, this contains a lot of unneeded information.
Abstract syntax tree

So, compilers often use an abstract syntax tree.

This is much more concise.

Abstract syntax trees (ASTs) are often used as an *il* between front end and back end.
Back end

\[ il \rightarrow \text{instruction selection} \rightarrow \text{register allocation} \rightarrow \text{machine code} \]

Responsibilities

- translate \( il \) into target machine code
- choose instructions for each \( il \) operation
- decide what to keep in registers at each point
- ensure conformance with system interfaces

*Automation has been less successful here*
Instruction Selection

- produce compact, fast code
- use available addressing modes
- pattern matching problem
  - \textit{ad hoc} techniques
  - tree pattern matching
  - string pattern matching
  - dynamic programming
Register allocation

Register Allocation

- have value in a register when used
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult
  \[ \Rightarrow \text{NP-complete for 1 or } k \text{ registers} \]

Modern allocators often use an analogy to graph coloring
Traditional three pass compiler

```
source code → front end → middle end → back end → machine code
```

Code Improvement

- analyzes and changes il
- goal is to reduce runtime
- must preserve values
Modern optimizers are usually built as a set of passes.

Typical passes

- discover & propagate constant values
- reduction of operator strength
- common subexpression elimination
- redundant computation elimination
- encode an idiom in some powerful instruction
- move computation to less frequently executed place (e.g., out of loops)