CENG 444

LANGUAGE PROCESSORS

Cem Bozşahin

Department of Computer Engineering
Middle East Technical University
Ankara

bozsahin@ceng.metu.edu.tr
GENERAL ARCHITECTURE OF LANGUAGE
INTERPRETATION

Alphabet (finite vocabulary)

Lexicon
(finitely characterizable tokens)

Grammar
(finitely characterizable sentences)

Well-formed programs
(meaningful computation
(source)'exec' code)
• Language processing (compiling/interpretation/translation) brings together Computer Architecture, OS, Formal Languages, Software Engineering and Programming Languages.

• Any computation can be visualized as the following process:
• Some virtual machines: Java Virtual Machine (JVM); FAM: Functional Abstract Machine; WAM: Warren’s Abstract Machine
USE OF COMPILERS IN AN ENVIRONMENT

front-end

LEXICAL ANALYSIS
LEXICAL ANALYSIS

SYNTACTIC ANALYSIS
SYNTACTIC ANALYSIS

SEMANTIC ANALYSIS
SEMANTIC ANALYSIS

IC OPTIMIZATION
IC OPTIMIZATION

I/O bound

CPU bound

I/O bound

CPU bound

source language
target language
relocatable code
absolute code

char stream
token stream
abstract representation of source program
intermediate code
intermediate code
target code

target code

RUN-TIME

TARGET CODE OPTIMIZER

TARGET CODE GEN

back-end
• Need for modularization: Portability, extendibility.

In many cases the stages are combined (single vs. multi-pass compilers)
• We will study compilers for imperative languages
• Different paradigms call for different compiler design; choice of intermediate code, compiling vs. interpretation; VM-based interpretation.
A Walk Through The Stages of Compilation

• XP: A language for arithmetic expressions

\[
\begin{align*}
S & \rightarrow \quad \text{ID} := \text{E;}
E & \rightarrow \quad \text{E} - \text{T} \mid \text{E} + \text{T} \mid \text{T}
T & \rightarrow \quad \text{T} * \text{F} \mid \text{T} / \text{F} \mid \text{F}
F & \rightarrow \quad \text{ID} \mid \text{NUM} \mid (\text{E})
\text{ID} & \rightarrow \quad \text{L} \ \text{ID}'
\text{ID}' & \rightarrow \quad (\text{L} \mid \text{D}) \ \text{ID}' \mid \epsilon
\text{NUM} & \rightarrow \quad \text{D} \ \text{NUM} \mid \text{D}
\end{align*}
\]

STAGE I: LEXICAL ANALYSIS. Tokenize the incoming stream of text.
Why separate lexical and syntactic analysis?
VAR1 := (B+C)*VAR2/(256−VAR3);
• Separation makes both stages simpler; The parser need not worry about internal structure of tokens, whitespace, comments etc.

• Usually, the grammar of a language is context-free, but the grammar of its tokens is regular. Use more efficient techniques.

• Machine-dependent I/O and alphabetical conventions can be localized.

<table>
<thead>
<tr>
<th>token types</th>
<th>patterns</th>
<th>lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>L(L</td>
<td>D)*</td>
</tr>
<tr>
<td>NUM</td>
<td>D^+</td>
<td>256</td>
</tr>
<tr>
<td>OP</td>
<td>(+</td>
<td>−</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>:=</td>
<td>:=</td>
</tr>
</tbody>
</table>
- VAR1 := (B+C)*VAR2 /(256-VAR3);

- ID ASSIGN DEL ID OP ID DEL ..... 

- A by-product of this stage is to start forming a table of meaning-bearing entities, called the symbol table.
Lex analyzer doesn’t know anything about the syntax of the language; it can fill ST with limited amount of information.

Symbol table is the most frequently accessed data structure in a compiler (lexical analyzer, parser, type checker, run-time system, optimizer etc.)

Need efficient insertion and search techniques for ST.
STAGING OF LEXICAL−SYNTACTIC ANALYSES

Producer

LEXICAL ANALYZER

next token

Consumer

SYNTACTIC ANALYZER

partial parse

ex: Fortran IF stmt
    IF(I,J)=3
STAGE II: SYNTACTIC ANALYSIS

Assign roles to tokens in parsing; set up symbol table; assign meanings to the use of tokens (syntax-directed translation).

- The meaning (semantics) of a program is what it does (computes).
A := B + C;

PARSE TREE for A := B + C

<table>
<thead>
<tr>
<th>rule</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S -&gt; ID := E;</td>
<td>output lvalue ID.name move</td>
</tr>
<tr>
<td>ID -&gt; ...</td>
<td>put ID in symbol table</td>
</tr>
<tr>
<td>T -&gt; F</td>
<td>T.val &lt;- F.val</td>
</tr>
<tr>
<td>E -&gt; E + T</td>
<td>output add</td>
</tr>
<tr>
<td>F -&gt; ID</td>
<td>output rvalue ID.name</td>
</tr>
<tr>
<td>F -&gt; NUM</td>
<td>output value NUM</td>
</tr>
</tbody>
</table>

ID := E ;
A E + T
T
F
ID
B
F
ID
C
• How to get semantic representation from the parse tree: associate semantic actions with rules.

• How do the attributes get instantiated?

  Depends on the parsing strategy.

  in Bottom-up parsing, the attributes are *synthesized* from value of child nodes.

  in top-down parsing, they are *inherited* by children.
• Choice of strategy also affects grammar. XP is left-recursive hence not very suitable for top-down parsing (re-write or use bottom-up).
• RECURSIVE-DESCENT PARSING: a top-down approach

Write a subprogram for each non-terminal in the grammar

For terminals, call the lexical analyzer

Flow of control shows the order of rule application

• ex: rewrite XP as a non-left-recursive grammar.

ex:  
S → ID := E ;
E → T E′
E′ → + TE′ | - T E′ | ε
match(T): returns true if next token is of type T

advance(): consumes the lookahead token

procedure S;
begin
  ID;
  if match(ASSIGN) then advance() else error();
  E;
  if match(STOP) then advance() else error();
end;

procedure ID;
begin
  if match(ID) then token:=advance();
  install_id(token);
end;

procedure E;
begin
  T;

procedure Eprime;
begin
  if match(OP) then {advance(); T; Eprime;}
  else /* no consumption */
end

Eprime;
end;
• BOTTOM-UP PARSING: obtain rightmost derivations *in reverse order*.

An algorithm to pick the right rule in derivations: LR parsing

A:=B+C;
ID:=B+C;
ID:=ID+C;
ID:=F+C;
ID:=T+C;
ID:=E+C;
ID:=E+ID;
ID:=E+F;
ID:=E+T;
ID:=E;
S
• STAGE III: SEMANTIC ANALYSIS

ex: type checking

\[
\begin{align*}
A &: int; \\
B &: real; \\
C &: = A / B;
\end{align*}
\]

need to generate code like

\[
\begin{align*}
T &: = \text{coerce}(A, real); \\
C &: = \text{divide}(T, B);
\end{align*}
\]
if (A+B) then ...

• STAGE IV: GENERATING INTERMEDIATE CODE (IC)

• choice of IC depend on source-target language and considerations.

1. Easy to translate into IC (assembly-like for imperative; lambda-calculus like for applicative langs)

2. Easy to obtain from source language (high-level assembler for imperative PLs; stack machines for arithmetic)
3. IC tends to be abstract three-address code (TAC) if RISC is the main target

IC tends to be two-address code if CISC is the main target

Architecture-independent virtual machines

- TAC: result := operand op operand

A:=(B+C-D)/2 translates to TAC

\[ t1 := B+C; \]
\[
\begin{align*}
t_2 & := t_1-D; \\
t_3 & := t_2/2; \\
A & := t_3;
\end{align*}
\]

- SAM: A stack machine for XP

fetch values of IDs from memory to stack (rvalue \(x\))

put values on stack (push \(v\))

put address of ID on stack (lvalue \(x\))
store in memory \textit{(move)}

operators

- ex: SAM code for $A := B + C - D + 10$;
• obtaining the SAM instructions during parsing (as a syntax-directed semantic action)
A := B + C;

S

ID := E ;

A E + T

T

F

ID

B
• From IC to Target Code (TC)

  Optimizations on IC: combine common subexpressions; eliminate dead code; fix loops; replace some calls with local go to’s

  Optimizations on TC: reduce memory fetch; maximize register use