CENG444: Language Processors (aka. Compilers)

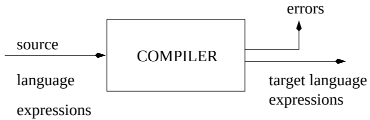
Cem Bozşahin

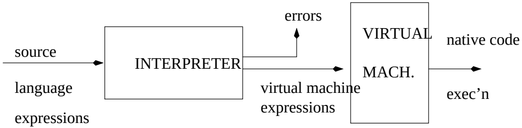
Cognitive Science Department, Informatics Institute,
Middle East Technical Univ. (ODTÜ), Ankara

October 1, 2018

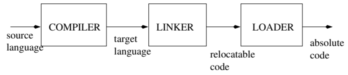
- Language processing (compiling/interpretation/translation) brings together
  - Computer Architecture,
  - OS,
  - Formal Languages,
  - Software Engineering
  - and Programming Languages.
  - And, as of lately, philosophy of computer science, such as Bozşahin (2018)

- Any computation can be visualized as the following process:

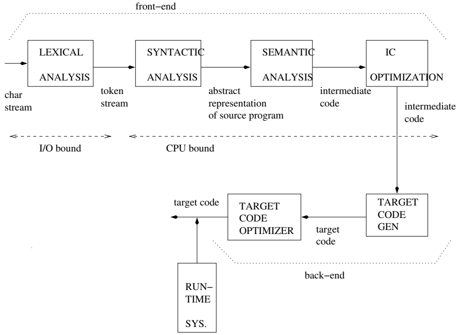




- Some virtual machines: Java Virtual Machine (JVM); FAM: Functional Abstract Machine; WAM: Warren's Abstract Machine



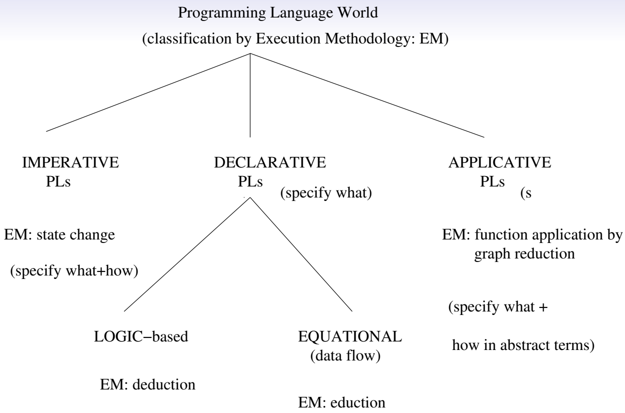
USE OF COMPILERS IN AN ENVIRONMENT



INSIDE THE BLACK BOX

- 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.

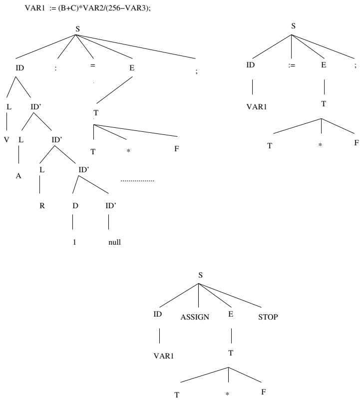
A Walk Through The Stages of Compilation

- XP: A language for arithmetic expressions

$$
\begin{array}{ll}
\mathsf{S} \rightarrow & \mathsf{ID} := \mathsf{E}; \\
\mathsf{E} \rightarrow & \mathsf{E} - \mathsf{T} \mid \mathsf{E} + \mathsf{T} \mid \mathsf{T} \\
\mathsf{T} \rightarrow & \mathsf{T} * \mathsf{F} \mid \mathsf{T} / \mathsf{F} \mid \mathsf{F} \\
\mathsf{F} \rightarrow & \mathsf{ID} \mid \mathsf{NUM} \mid (\mathsf{E}) \\
\mathsf{ID} \rightarrow & \mathsf{L}\ \mathsf{ID}' \\
\mathsf{ID}' \rightarrow & (\mathsf{L} \mid \mathsf{D})\ \mathsf{ID}' \mid \varepsilon \\
\mathsf{NUM} \rightarrow & \mathsf{D}\ \mathsf{NUM} \mid \mathsf{D}
\end{array}
$$

STAGE I: LEXICAL ANALYSIS. Tokenize the incoming stream of text.

Why separate lexical and syntactic analysis?



- 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.

token types	patterns	lexemes
ID	$L(L\|D)^*$	var1, b, abc5rd
NUM	D^+	256
OP	$(+\mid-\mid*\mid/)$	$+$
ASSIGN	:=	:=

- VAR1 := (B+C)*VAR2 /(256-VAR3);

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

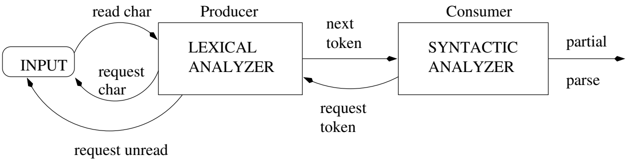
- A by-product of this stage is to start forming a table of meaning-bearing entities, called *the symbol table*.

- 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

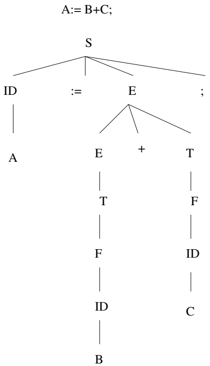


ex: Fortran IF stmt IF(I,J)=3

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).



rule	action
S –> ID := E;	output lvalue ID.name move
ID –> ...	put ID in symbol table
T –> F	T.val <– F.val
E –> E + T	output add
F –> ID	output rvalue ID.name
F –> NUM	output value NUM

- 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:
$$\begin{aligned} S &\rightarrow \quad ID := E \; ; \\ E &\rightarrow \quad T\ E' \\ E' &\rightarrow \quad +\ TE' \mid -\ T\ E' \mid \varepsilon \end{aligned}$$

```
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;
   Eprime;
end;

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

- BOTTOM-UP PARSING: obtain rightmost derivations *in reverse order*.

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

- XP: A language for arithmetic expressions

$$
\begin{array}{ll}
\text{S} \rightarrow & \text{ID} := \text{E;} \\
\text{E} \rightarrow & \text{E - T} \mid \text{E + T} \mid \text{T} \\
\text{T} \rightarrow & \text{T * F} \mid \text{T / F} \mid \text{F} \\
\text{F} \rightarrow & \text{ID} \mid \text{NUM} \mid \text{(E)} \\
\text{ID} \rightarrow & \text{L ID}' \\
\text{ID}' \rightarrow & \text{(L} \mid \text{D) ID}' \mid \varepsilon \\
\text{NUM} \rightarrow & \text{D NUM} \mid \text{D}
\end{array}
$$

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

```
A: int;
B: real;

C:= A/B;
```

  need to generate code like

```
T:=coerce(A, real);
C:=divide(T,B);
```

- 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

$$
\begin{aligned}
t1 &:= B+C\,; \\
t2 &:= t1-D\,; \\
t3 &:= t2\,/\,2\,; \\
A &:= t3\,;
\end{aligned}
$$

- 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`)

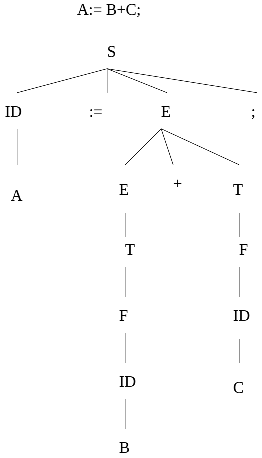
  store in memory (`move`)

  operators

ex: SAM code for A:=B+C-D+10;

```
rvalue B
rvalue C
ADD
rvalue D
SUB
push 10
ADD
lvalue A
move
```

- obtaining the SAM instructions during parsing (as a syntax-directed semantic action)



- From IC to Target Code (TC)

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

  Optimizations on TC: reduce memory fetch; maximize register use

Bozşahin, C. (2018). Computers aren't syntax all the way down or content all the way up. *Minds and Machines 28*(3), 543–567.