1. Introduction

Eva Rose Kristoffer Rose

NYU Courant Institute

Compiler Construction (CSCI-GA.2130-001)

http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/lecture-1.pdf

September 4, 2014

Administrivia

Web Page: *cs.nyu.edu/courses/fall14/CSCI-GA.2130-001*

Who: Eva Rose and Kristoffer Rose

Grader: Weicheng Ma

Class: Thursdays, 5:10-7 pm, CIWW 312

Office Hours: Thursdays, 4-5 pm, CIWW 312

Grading: Projects (45%), Homework (15%), Midterm (15%), Final (25%)

Please check your e-mail this week for contact information!

Why study compilers?

What is a compiler, what does it do?

Compiler

source program → COMPILER → target program

- source programs: typically high level,
- target programs: typically assembler or object/machine code,
- compiler implementations, e.g. C, ML, Python, Java ...

Advantages of compilers:

- allow programming at an understandable abstraction level,
- allow programs to be written in machine-independent languages,
- help in verifying software and error reporting,
- help code optimization.

Historical background:

- Grace Hopper coins the concept and writes the first compiler in 1952,
- John W. Backus presents the first formally based compiler (FORTRAN) in 1957,
- Frances E. Allen (Turing award '06), John Cocke introduce most of the abstract concepts used in compiler optimization and parallel compilers today,

Meta-language: a language to talk about another language

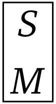
Essential Language Processors

Interpreter: a program (written in a meta language) for executing another program.

Compiler: a program (written in a meta language) that translates a program into an equivalent program.

Interpreters

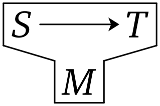
Interpreter diagrams, I-diagrams:



- *S* – *Source* language
- *M* – *Meta* or *Implementation* language

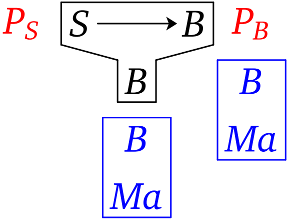
Compilers

Compiler diagrams, T-diagrams:



- *S* – compiled *Source* language
- *T* – generated *Target* language
- *M* – *Meta* or *Implementation* language

Hybrid: The Java Compiler



- S – compiled *Source* language
- B – Intermediate *Bytecode* language
- Ma – actual *Machine* language

Language-Processing System

Preprocessor: expands “macros” and combines source program modules.

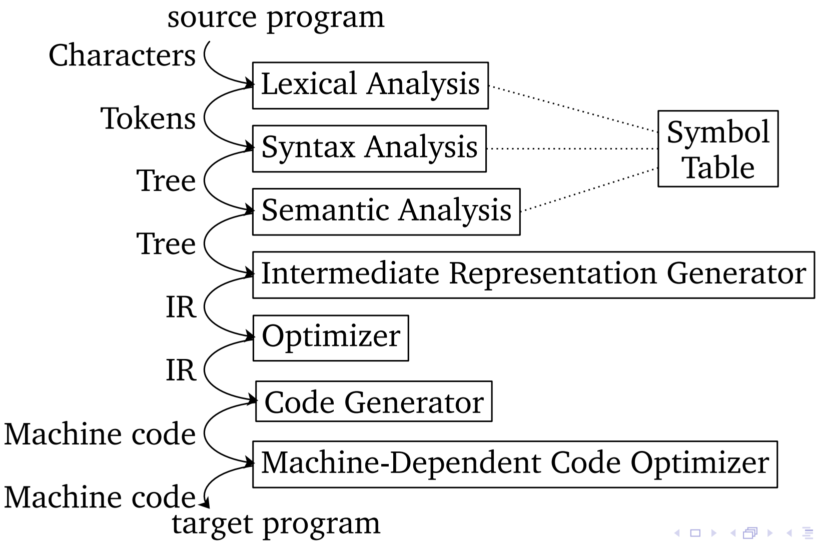
Compiler: translates source language to symbolic (assembler) machine code.

Assembler: translates symbolic machine code to relocatable binary code.

Linker: resolves links to library files and other relocatable object files.

Loader: combines executable object files into memory.

Phases


source program
Characters
Lexical Analysis
Tokens
Syntax Analysis
Tree
Semantic Analysis
Tree
Intermediate Representation Generator
IR
Optimizer
IR
Code Generator
Machine code
Machine-Dependent Code Optimizer
Machine code
target program
Symbol Table

Example

Perceived as *stream of characters*:

```
position = initial + rate * 60
```

Note: the undefined variables are assumed floating points.

Lexemes

From Linguistics we have that a *lexeme* is the *smallest meaningful entity* of a language.

The lexemes here: `position`, =, `initial`, +, `rate`, *, and `60`.

Lexical Analysis

```
position = initial + rate * 60
```

scanned into list of *tokens*, one for each *lexeme*:

$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle \mathbf{num}, 60\rangle$

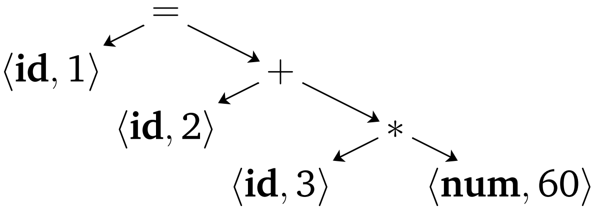
1	`position`
2	`initial`
3	`rate`

Syntax Analysis

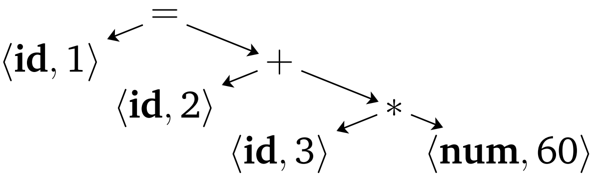
$$\langle \mathbf{id}, 1\rangle \quad \langle = \rangle \quad \langle \mathbf{id}, 2\rangle \quad \langle + \rangle \quad \langle \mathbf{id}, 3\rangle \quad \langle * \rangle \quad \langle \mathbf{num}, 60\rangle$$

1	`position`
2	`initial`
3	`rate`

parsed into *syntax tree* (precedence):

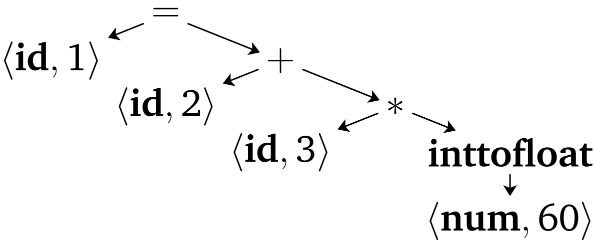


Semantic Analysis

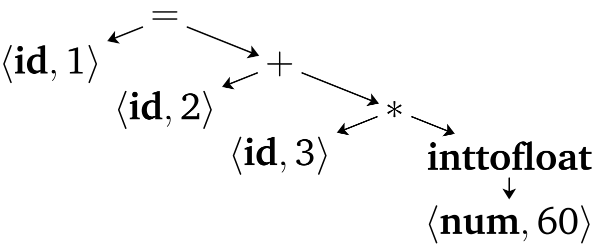


1	position
2	initial
3	rate

enriched with semantic information (explicit type conversion):



Intermediate Representation Generation



1	position
2	initial
3	rate

translated to intermediate code:

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Optimization

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

optimized to

```
t1 = id3 * 60.0
id1 = id2 + t1
```

1	position
2	initial
3	rate

Code Generation

```
t1 = id3 * 60.0
id1 = id2 + t1
```

1	position
2	initial
3	rate

generates

```
LDF  R2,  id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Compiler-construction tools

Commonly used tools:

- scanner generators: token description $\rightarrow$ lexical analyser,
- parser generators: grammar $\rightarrow$ syntax analyser,
- syntax-directed translation engines: syntax tree $\rightarrow$ IR,
- code-generator generators: translation rules $\rightarrow$ code generator,
- Data-flow engines: data-flow information analyzers.

Compiler-generation: integrated set of the above.

HACS

HACS is a compiler generator:

- for most compiler phases,
- from formal specifications,
- formalisms are grammars and syntax-directed definitions.

HACS

The HACS compiler generator:

- stands for Higher-order Attribute Contraction Schemes,
- created by Kristoffer Rose,
- part of the open source "CRSX" project (*crsx.org*),
- commercially in use by IBM.

Compiling with HACS

Compiling source program (term) from fig. 1.7 with command:

```
$ ./first.run --action=Compile \
  --term="{initial:=1; rate:=1.0; position:=initial+rate*60;}"
```

Compiling with HACS

... outputs the target program (fig.1.7):

```
LDF T , #1
STF initial , T
LDF T_40 , #1.0
STF rate , T_40
LDF T_1 , initial
LDF T_!_90 , rate
LDF T_2 , #60
MULF T_2_84 , T_1_90 , T_2
ADDF T_25 , T_1 , T_2_84
STF  position , T_25
```

Lexical Analyzer in HACS

First step of fig. 1.7:

Formalism at play: *regular expressions*

```
space [ \t\n] ;

token Int   | ⟨Digit⟩+ ;
token Float | ⟨Int⟩ "." ⟨Int⟩ ;
token Id    | ⟨Lower⟩+ ('_'? ⟨Int⟩)? ;

token fragment Digit | [0−9] ;
token fragment Lower | [a−z] ;
```

Lexical Analyzer in HACS

A lexeme (term) gets lexically analysed (sort) with the command:

```
$ ./first.run --sort=Float --term=34.56
```

.. and outputs recognized token:

```
34.56
```

.. or reports error message (with `--sort=Int`):

```
.. parse error .. Encountered <T_FLOAT> "34.56" at line 1, column 1
was expecting one of:  <T_INT> ..
```

Syntax Analyzer in HACS

Second step of fig. 1.7:

Formalism at play: *context-free grammars*

```
sort Stat  | ⟦ ⟨Name⟩ := ⟨Exp⟩ ; ⟧| ⟦{ ⟨Stat∗⟩ } ⟧;

sort Exp   | ⟦ ⟨Exp@1⟩ + ⟨Exp@2⟩ ⟧@1
           | ⟦ ⟨Exp@2⟩ ∗ ⟨Exp@3⟩ ⟧@2
           | ⟦ ⟨Int⟩ ⟧@3
           | ⟦ ⟨Float⟩ ⟧@3
           | ⟦ ⟨Name⟩ ⟧@3
           | sugar ⟦(⟨Exp@1#⟩) ⟧@3 →# ;

sort Name | symbol ⟦⟨Id⟩ ⟧;
```

Syntax analyser with HACS

An expression (term) gets syntactically analysed/parsed (action) with the command:

```
$ ./first --sort=Exp --term="(2+(3*(4+5)))"
```

.. and outputs the grammar-checked expression:

```
2 + 3 * ( 4 + 5 )
```

.. or reports error message...

Intermediate Code Generation

Fourth step of fig. 1.7:

Formalism: Syntax directed translation schemes

sort IntermediateCode | **scheme** ⟦Generate ⟨Temp⟩ ⟨Exp⟩ ⟧;

...

```
⟦ Generate t ⟨Exp#1⟩ + ⟨Exp#2⟩ ⟧
  → ⟦
       {Generate t1 ⟨Exp#1⟩}
       {Generate t2 ⟨Exp#2⟩}
       t = t1 + t2;
```