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
1. Administrivia
2. Why study compilers?
3. Language Processors
4. Structure of a Compiler
5. Compiler generation tool: HACS
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?
source program $\rightarrow$ COMPILER $\rightarrow$ 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,
1. Administrivia

2. Why study compilers?

3. Language Processors

4. Structure of a Compiler

5. Compiler generation tool: HACS
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.
1. Introduction

1. Administrivia

2. Why study compilers?

3. Language Processors

4. Structure of a Compiler

5. Compiler generation tool: HACS
**Phases**

source program

Characters

Lexical Analysis

Syntax Analysis

Semantic Analysis

Symbol Table

Syntax Analysis

Syntax Analysis

Syntax Analysis

Tree

Tree

Tree

Intermediate Representation Generator

Optimizer

Code Generator

Machine code

Machine code

Target program
Example

Perceived as *stream of characters*:

\[
\text{position} = \text{initial} + \text{rate} \times 60
\]

Note: the undefined variables are assumed floating points.
Lexemes

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

Lexical Analysis

\[ \text{position} = \text{initial} + \text{rate} \times 60 \]

scanned into list of tokens, one for each lexeme:

\[
\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle \text{num}, 60 \rangle
\]

\[
\begin{array}{|c|}
\hline
1 & \text{position} \\
2 & \text{initial} \\
3 & \text{rate} \\
\hline
\end{array}
\]
Syntax Analysis

\[ \langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle \text{num}, 60 \rangle \]

parsed into syntax tree (precedence):

\[
\begin{array}{c}
\text{id, 1} \\
\text{id, 2} \\
\text{id, 3} \\
\text{num, 60}
\end{array}
\]

\[
\begin{array}{c}
\text{1} \quad \text{position} \\
\text{2} \quad \text{initial} \\
\text{3} \quad \text{rate}
\end{array}
\]
Semantic Analysis

enriched with semantic information (explicit type conversion):

\[
\begin{align*}
\langle \text{id}, 1 \rangle & \quad = \quad + \\
\langle \text{id}, 2 \rangle & \quad \ast \\
\langle \text{id}, 3 \rangle & \quad \rightarrow \\
\langle \text{num}, 60 \rangle & \\
\end{align*}
\]

\[
\begin{align*}
\langle \text{id}, 1 \rangle & \quad = \quad + \\
\langle \text{id}, 2 \rangle & \quad \ast \\
\langle \text{id}, 3 \rangle & \quad \rightarrow \\
\langle \text{int} \rightarrow \text{float} \rangle & \quad \rightarrow \\
\langle \text{num}, 60 \rangle & \\
\end{align*}
\]
Intermediate Representation Generation

\[
\langle \text{id}, 1 \rangle \xrightarrow{=} \langle \text{id}, 2 \rangle + \langle \text{id}, 3 \rangle \xrightarrow{\ast} \text{inttofloat} \downarrow \langle \text{num}, 60 \rangle
\]

translated to intermediate code:

1. \( t1 = \text{inttofloat}(60) \)
2. \( t2 = \text{id}3 \times t1 \)
3. \( t3 = \text{id}2 + t2 \)
4. \( \text{id}1 = t3 \)
Optimization

\begin{verbatim}
1 \hspace{1em} t1 = inttofloat(60)
2         t2 = id3 * t1
3         t3 = id2 + t2
4         id1 = t3

optimized to

1 \hspace{1em} t1 = id3 * 60.0
2         id1 = id2 + t1
\end{verbatim}
Code Generation

\[
\begin{align*}
1. & \quad t1 = id3 \times 60.0 \\
2. & \quad id1 = id2 + t1
\end{align*}
\]

generates

\[
\begin{align*}
1. & \quad \text{LDF R2, id3} \\
2. & \quad \text{MULF R2, R2, #60.0} \\
3. & \quad \text{LDF R1, id2} \\
4. & \quad \text{ADDF R1, R1, R2} \\
5. & \quad \text{STF id1, R1}
\end{align*}
\]
1. Administrivia
2. Why study compilers?
3. Language Processors
4. Structure of a Compiler
5. Compiler generation tool: HACS
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 is a compiler generator:

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

```plaintext
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: \textit{context-free grammars}

\begin{verbatim}
sort Stat | [ [\langle\text{Name}\rangle := \langle\text{Exp}\rangle ; ] ] ] ;

sort Exp | [ [\langle\text{Exp}@1\rangle + \langle\text{Exp}@2\rangle ] ]@1
| [ [\langle\text{Exp}@2\rangle * \langle\text{Exp}@3\rangle ] ]@2
| [ [\langle\text{Int}\rangle ] ]@3
| [ [\langle\text{Float}\rangle ] ]@3
| [ [\langle\text{Name}\rangle ] ]@3
| sugar [([\langle\text{Exp}@1\#\rangle ) ]@3 →# ;

sort Name | symbol [\langle\text{Id}\rangle ];
\end{verbatim}
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

\[
\text{sort IntermediateCode} \mid \text{scheme } \left[ \text{Generate } \langle \text{Temp} \rangle \langle \text{Exp} \rangle \right];
\]

\[
\left[ \text{Generate } t \langle \text{Exp}\#1 \rangle + \langle \text{Exp}\#2 \rangle \right]
\rightarrow \left[
\begin{align*}
\{ & \text{Generate } t_1 \langle \text{Exp}\#1 \rangle \\
\{ & \text{Generate } t_2 \langle \text{Exp}\#2 \rangle \\
t & = t_1 + t_2;
\end{align*}
\right]
\]