Compiler Construction
Introduction[1]

Kristoffer H. Rose Eva Rose

NYU Courant Institute

January 27, 2014

[1] ALSU §§1.1–1.2

1. Administrivia
2. Language Processors
3. Structure of a Compiler
4. HACS

Administrivia

Web Page:	*cs.nyu.edu/courses/spring14/CSCI-GA.2130-001*
Who:	Kristoffer Rose, Eva Rose
Class:	Mondays, 7.10-9pm, CIWW 202
Office Hours:	Mondays, 5-6pm, CIWW 328
Grader:	Weicheng Ma
Grading:	Homework (10%), Midterm (10%), Project (50%), Final (30%)

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

1 Administrivia

2 Language Processors

3 Structure of a Compiler

4 HACS

Language Processors

Interpreter: Processor that reads a program and executes it on input data.

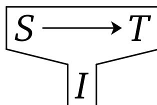
Compiler: Processor that reads a program and writes an equivalent program in another language.

Interpreter

S
I

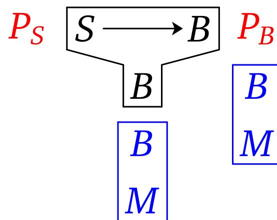
- S – interpreted *Source* language
- I – *Implementation* language

Compiler



- S – compiled *Source* language
- T – generated *Target* language
- I – *Implementation* language

Hybrid



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

Language-Processing System

Preprocessor:	Expand “macros.”
Compiler:	Translate source language to symbolic machine code.
Assembler:	Translate symbolic machine code to relocatable binary code.
Linker:	Resolve links to library files.

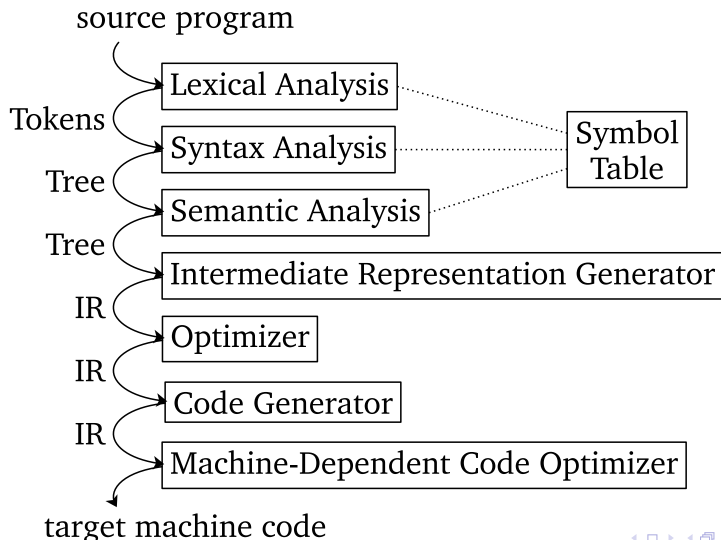
1 Administrivia

2 Language Processors

3 Structure of a Compiler

4 HACS

Phases



Example

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

Note: undefined variables are assumed floating point.

Lexical Analysis

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

scanned into list of *tokens*:

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

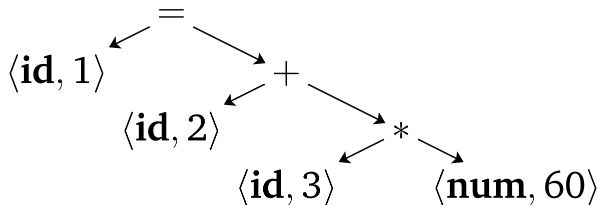
1	position
2	initial
3	rate

Syntax Analysis

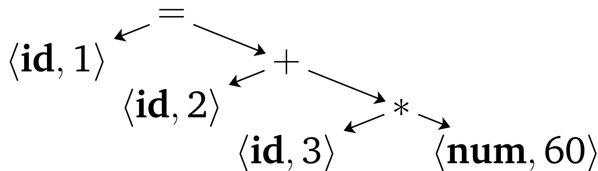
$$\langle \mathbf{id}, 1\rangle \quad \langle = \rangle \quad \langle \mathbf{id}, 2\rangle \quad \langle + \rangle \quad \langle \mathbf{id}, 3\rangle \quad \langle * \rangle \quad \langle \mathbf{num}, 60\rangle$$

1	position
2	initial
3	rate

parsed into *parse tree*:

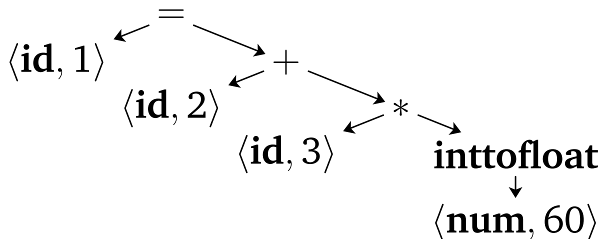


Semantic Analysis

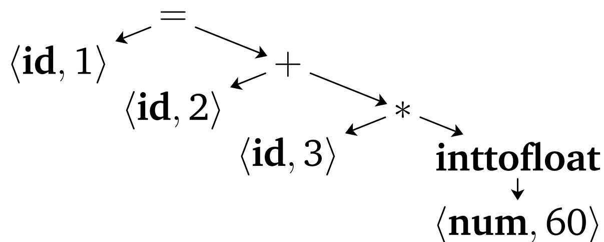


1	position
2	initial
3	rate

enriched with semantic information:



Intermediate Representation Generator



1	position
2	initial
3	rate

translated to intermediate code:

```
1    t1 = inttofloat(60)
2    t2 = id3 * t1
3    t3 = id2 + t2
4    id1 = t3
```

Optimizer

```
1   t1 = inttofloat(60)
2   t2 = id3 * t1
3   t3 = id2 + t2
4   id1 = t3
```

optimized to

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

1	position
2	initial
3	rate

Code Generator

1	position
2	initial
3	rate

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

generates

```
1        LDF  R2,  id3
2        MULF R2, R2, #60.0
3        LDF  R1, id2
4        ADDF R1, R1, R2
5        STF  id1, R1
```

1 Administrivia

2 Language Processors

3 Structure of a Compiler

4 HACS

HACS

- Stands for Higher-order Attribute Contraction Schemes.
- Builds compiler from formal specification of compiler phases.
- Formalisms are grammars and syntax-directed translations.
- Part of open source “CRSX” project (*crsx.sf.net*).

Lexical Analysis

```
space [ \t\n] ;

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

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

Syntax Analysis

```
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#⟩) ⟧@3 → # ;

sort Name    | symbol ⟦ ⟨Id⟩ ⟧ ;
```

Semantic Analysis (type sort and synthesis)

```
sort Type | Int | Float ;

| scheme Unif(Type,Type) ;
Unif(Int, Int) → Int;
Unif(#t1, Float) → Float;
Unif(Float, #t2) → Float;

attribute ↑t(Type);
sort Exp; ↑t;

⟦ (⟨Exp#1 ↑t(#t1)⟩ + ⟨Exp#2 ↑t(#t2)⟩) ⟧ ↑t(Unif(#t1,#t2));
⟦ (⟨Exp#1 ↑t(#t1)⟩ * ⟨Exp#2 ↑t(#t2)⟩) ⟧ ↑t(Unif(#t1,#t2));
⟦ ⟨Int#⟩ ⟧ ↑t(Int);
⟦ ⟨Float#⟩ ⟧ ↑t(Float);
```

Semantic Analysis (statement analysis)

```
attribute ↓e{Name:Type};

sort Stat | scheme ⟦ TA ⟨Stat⟩ ⟧ ↓e ;

⟦ TA ⟨Name id⟩ := ⟨Exp#2⟩; ⟧ → ⟦ ⟨Name id⟩ := TA ⟨Exp#2⟩; ⟧;
⟦ TA {} ⟧ → ⟦{}⟧;
⟦ TA { ⟨Name id⟩ := ⟨Exp#2⟩; ⟨Stat*#3⟩ } ⟧
  → ⟦ TA2 { ⟨Name id⟩ := TA ⟨Exp#2⟩; ⟨Stat*#3⟩ } ⟧;
{
  | scheme ⟦ TA2 ⟨Stat⟩ ⟧ ↓e;
  ⟦ TA2 { ⟨Name id⟩ := ⟨Exp#2 ↑t(#t2)⟩; ⟨Stat*#3⟩ } ⟧ →
    ⟦ { ⟨Name id⟩ := ⟨Exp#2⟩; ⟨Stat ⟦TA {⟨Stat*#3⟩}⟧ ↓e{⟦id⟧:#t2}⟩ } ⟧;
}
⟦ TA { {⟨Stat*#1⟩} ⟨Stat*#2⟩ } ⟧ → ⟦ { TA {⟨Stat*#1⟩} TA {⟨Stat*#2⟩} } ⟧;
```

Semantic Analysis (expression analysis)

```
sort Exp | scheme ⟦ TA ⟨Exp⟩ ⟧ ↓e ;

⟦ TA id ⟧ ↓e{⟦id⟧ : #t} → ⟦ id ⟧ ↑t(#t);
⟦ TA id ⟧ ↓e{¬⟦id⟧} → error⟦Undefined identifier ⟨id⟩⟧;

⟦ TA ⟨Int#⟩ ⟧ → ⟦ ⟨Int#⟩ ⟧;
⟦ TA ⟨Float#⟩ ⟧ → ⟦ ⟨Float#⟩ ⟧;
⟦ TA (⟨Exp#1⟩ + ⟨Exp#2⟩) ⟧ → ⟦ (TA ⟨Exp#1⟩) + (TA ⟨Exp#2⟩) ⟧;
⟦ TA (⟨Exp#1⟩ * ⟨Exp#2⟩) ⟧ → ⟦ (TA ⟨Exp#1⟩) * (TA ⟨Exp#2⟩) ⟧;
```

Intermediate Code Generation (IR syntax)

```
token T | T ('_' ⟨Int⟩)? ; // temporary

sort I_Progr | ⟦⟨I_Instr⟩ ⟨I_Progr⟩⟧ | ⟦⟧ ;

sort I_Instr | ⟦⟨Tmp⟩ = ⟨I_Arg⟩ + ⟨I_Arg⟩;¶⟧
             | ⟦⟨Tmp⟩ = ⟨I_Arg⟩ * ⟨I_Arg⟩;¶⟧
               | ⟦⟨Tmp⟩ = ⟨I_Arg⟩;¶⟧
               | ⟦⟨Name⟩ = ⟨Tmp⟩;¶⟧
     ;

sort I_Arg | ⟦⟨Name⟩⟧ | ⟦⟨Float⟩⟧ | ⟦⟨Int⟩⟧ | ⟦⟨Tmp⟩⟧ ;

sort Tmp | symbol ⟦ ⟨T⟩ ⟧ ;
```

Intermediate Code Generation (statements)

```
attribute ↓TmpType{Tmp:Type} ;

sort I_Progr ;

| scheme ⟦ ICG ⟨Stat⟩ ⟧ ↓TmpType ;
⟦ ICG id := ⟨Exp#2 ↑t(#t2)⟩; ⟧ → ⟦ { ⟨I_Progr ⟦ICGExp T ⟨Exp#2⟩⟧ ↓TmpT
⟦ ICG { } ⟧ → ⟦ ⟧;
⟦ ICG { ⟨Stat#s⟩ ⟨Stat*#ss⟩ } ⟧ → ⟦ { ICG ⟨Stat#s⟩ } ICG { ⟨Stat*#ss⟩
```

Intermediate Code Generator (expressions)

```
| scheme ⟦ ICGExp ⟨Tmp⟩ ⟨Exp⟩ ⟧ ;

⟦ ICGExp T ⟨Int#1⟩ ⟧ → ⟦ T = ⟨Int#1⟩; ⟧ ;
⟦ ICGExp T ⟨Float#1⟩ ⟧ → ⟦ T = ⟨Float#1⟩; ⟧ ;
⟦ ICGExp T id ⟧ → ⟦ T = id; ⟧ ;

⟦ ICGExp T ⟨Exp#1⟩ + ⟨Exp#2⟩ ⟧
  → ⟦ {ICGExp T_1 ⟨Exp#1⟩} {ICGExp T_2 ⟨Exp#2⟩} T = T_1 + T_2; ⟧ ;

⟦ ICGExp T ⟨Exp#1⟩ * ⟨Exp#2⟩ ⟧
  → ⟦ {ICGExp T_1 ⟨Exp#1⟩} {ICGExp T_2 ⟨Exp#2⟩} T = T_1 * T_2; ⟧ ;

| scheme ⟦ {⟨I_Progr⟩} ⟨I_Progr⟩ ⟧; // Flattening helper
⟦ {} ⟨I_Progr#3⟩ ⟧ → #3 ;
⟦ {⟨I_Instr#1⟩ ⟨I_Progr#2⟩} ⟨I_Progr#3⟩ ⟧
  → ⟦ ⟨I_Instr#1⟩ {⟨I_Progr#2⟩} ⟨I_Progr#3⟩ ⟧;
```

Code Generator (syntax)

```
sort A_Progr | ⟦ ⟨A_Instr⟩ ⟨A_Progr⟩ ⟧ | ⟦⟧ ;

sort A_Instr | ⟦ LDF ⟨Tmp⟩, ⟨A_Arg⟩¶⟧
             | ⟦ STF ⟨Name⟩, ⟨Tmp⟩¶⟧
             | ⟦ ADDF ⟨A_Arg⟩, ⟨A_Arg⟩, ⟨A_Arg⟩¶⟧
             | ⟦ MULF ⟨A_Arg⟩, ⟨A_Arg⟩, ⟨A_Arg⟩¶⟧
     ;

sort A_Arg | ⟦ #⟨Float⟩ ⟧ | ⟦ #⟨Int⟩ ⟧ | ⟦ ⟨Name⟩ ⟧ | ⟦ ⟨Tmp⟩ ⟧ ;
```

Code Generator (program)

```
sort A_Progr | scheme ⟦ CG ⟨I_Progr⟩ ⟧ ;

⟦ CG ⟧ → ⟦⟧ ;

⟦ CG T = ⟨I_Arg#1⟩ + ⟨I_Arg#2⟩ ; ⟨I_Progr#⟩ ⟧
  → ⟦ ADDF T, [⟨I_Arg#1⟩], [⟨I_Arg#2⟩] CG ⟨I_Progr#⟩ ⟧ ;

⟦ CG T = ⟨I_Arg#1⟩ * ⟨I_Arg#2⟩ ; ⟨I_Progr#⟩ ⟧
  → ⟦ MULF T, [⟨I_Arg#1⟩], [⟨I_Arg#2⟩] CG ⟨I_Progr#⟩ ⟧ ;

⟦ CG T = ⟨I_Arg#1⟩ ; ⟨I_Progr#⟩ ⟧
  → ⟦ LDF T, [⟨I_Arg#1⟩] CG ⟨I_Progr#⟩ ⟧ ;

⟦ CG name = T ; ⟨I_Progr#⟩ ⟧ → ⟦ STF name, T CG ⟨I_Progr#⟩ ⟧ ;
```

Code Generator (arguments)

```
sort A_Arg ;

| scheme ⟦ [⟨I_Arg⟩] ⟧ ;
⟦ [T] ⟧ → ⟦ T ⟧ ;
⟦ [name] ⟧ → ⟦ name ⟧ ;
⟦ [⟨Float#1⟩] ⟧ → ⟦ #⟨Float#1⟩ ⟧ ;
⟦ [⟨Int#1⟩] ⟧ → ⟦ #⟨Int#1⟩ ⟧ ;
```

Main

```
// $Id: first.hx,v 1.16 2014/01/06 03:11:28 krisrose Exp $
// First HACS sample compiler (inspired by Dragonbook Fig. 1.7).
//
module "net.sf.crsx.samples.gentle.First" {

  ...

  sort A_Progr | scheme Compile(Stat);
  Compile(#) → ⟦ CG ICG TA ⟨Stat#1⟩ ⟧ ;
}
```