Compilers

Prof. Samuel P. Midkiff
smidkiff@purdue.edu
Office - EE 334A

ECE 468 Fall 2009

http://www.ece.purdue.edu/~smidkiff/ece468f09

OFFICE HOURS, COURSE COMMUNICATION

- Put 495S in the subject line, and I will try and do the same with mail I send
- Office hours are currently Tuesday 10:30 – 11:30 and Thursday 2-3. These may change.
- The course webpage and newsgroup will be where you should look for announcements, etc. (search on sam midkiff to find my webpage, course page link is at the bottom)

A SHORT HISTORY OF COMPUTING

- The earliest analog computer was built around 100 BC
- The Difference Engine was developed in the mid-1800s
- Early digital computers were programmed using punched cards that were repeatedly read, physically reconfiguring the computer, and programming in binary
  + Not all were Turing complete
TWO CONCEPTUAL BREAKTHROUGHS

- Stored program computer
  - Von Neumann, program is not modifiable
  - Harvard architecture, program in data storage
- Programming language and compiler

PROGRAMMING LANGUAGES & COMPILERS

- Programming languages to write programs
  + Hopper's A-0 handled link-editing chores, passing parameters to subroutines
  + Assemblers, i.e. IBM's Autocoder, symbolic names for storage, addresses, instruction mnemonics
  + Programming languages and compilers, Fortran in 1957, COBOL 1960
  + Algol 60 – reserved words, dynamic arrays, user defined data types
  + Lisp, 1962, first self-hosting language
  + Simula, 1967, objects, classes, subclasses, virtual methods, garbage collection
  + Pascal, 1968, first language with strong typing, managed pointers
  + C, 1972, unmanaged pointers, weak typing and casts, could write most of an OS in it.
  + Smalltalk-80, first OO language called an OO language (Alan Kay, Xerox Parc)
  + Java, 1990s – return of interpretation, strong typing, GC in the mainstream

COMPILERS ARE TRANSLATORS

- Fortran
- C
- C++
- Java
- Text processing language
- HTML/XML
- Command & Scripting Languages
- Natural language
- Domain specific languages

- Machine code
- Virtual machine code
- Transformed source code
- Augmented source code
- Low-level commands
- Semantic components
- Another language

IMPORTANCE OF COMPILERS

- Machines get more complicated
  + Pipelined processors, delay slots, performance dependences on instruction order and data fetch sequences make assembly language programming hard
- Applications get more complicated
  + IBM OS/360, ~6 million lines of code nearly destroyed IBM
  + Windows XP, 45+ million lines of code, barely fazed Microsoft financially
  + Both were delivered late …
WHO IS BETTER – MAN OR MACHINE?

- On short instruction sequences (tens of lines of code) skilled humans tend to be better
- Over a few hundred lines, compilers are better (don’t get bored with myriad details)
- Algorithm design swamps both of these
- Very first (Fortran) compiler in the 1950s did within a few percent of humans (see *The Fortran I Compiler*, David Padua, IEEE CSE)

FOUR TRANSLATION SEQUENCES

1. High level language translated to assembly language of some computer
2. High level language translated to machine independent “byte code” (Java), “p-code” (Turbo Pascal) or “CIL” (Microsoft) which is later compiled further or interpreted
3. Machine independent byte code or CIL translated to native machine instructions
4. HLL language to another HLL (especially for domain specific languages and research compilers)

We will focus on the first of these since the other two are similar

ASSEMBLY CODE AND ASSEMBLERS

- Assemblers are often used at the compiler back-end.
- Assemblers are low-level translators.
- They are machine-specific, and perform mostly 1:1 translation between mnemonics and machine code, except:
  - symbolic names for storage locations
  - program locations (branch, subroutine calls)
  - variable names
  - macros

COMPILERS AND INTERPRETERS

- Compilers sometimes generate code to be executed by an interpreter rather than generating native code or assembly code
- E.g. *javac* generates bytecode. Forth generates threaded code, and Borland’s Turbo Pascal generated P-code.
- Only the interpreter has to be ported to get portability – usually easier than porting a compiler code generator (or backend).
Consider the expression \( y = a + b \times c \)

Can be compiled into:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 push b</td>
<td>Push b into the stack</td>
</tr>
<tr>
<td>03 push c</td>
<td>Push c into the stack</td>
</tr>
<tr>
<td>06 mul</td>
<td>Multiply the top two elements of the stack</td>
</tr>
<tr>
<td>07 push a</td>
<td>Push a into the stack</td>
</tr>
<tr>
<td>10 add</td>
<td>Add the top two elements of the stack</td>
</tr>
<tr>
<td>11 pop y</td>
<td>Pop the element from the stack and store it in y</td>
</tr>
</tbody>
</table>

Each instruction corresponds to an instruction in the machine's language.

Note that no real machine exists for this code. Will execute by interpretation.

**INTERPRETATION GOOD POINTS:**

- "execution" is immediate
- elaborate error and other runtime checking is possible
- can change program on-the-fly. E.g., switch libraries, dynamic change of data types
- machine independence – isolates programs from real machine characteristics. E.g., Java byte code
- Code is often smaller (good for embedded)

**INTERPRETER BAD POINTS**

- Interpretation is slower than executing even naively compiled code by 2x – 3x times
- The interpreter needs to reside in memory
  - However, interpreted codes, e.g. Java byte-codes, can be denser than other native (machine) code.
  - Interpreter is typically much smaller than a dynamic (just-in-time) compiler.
**Dynamic (Just-in-Time) Compilers**

- Run in the same environment as interpreter
- Compile, on-the-fly, byte code, p-code, etc.
- Java the most notable example of this technique
- Microsoft .Net infrastructure will use similar techniques (CIL, Common Intermediate Language)
- Compiler structure similar to traditional C, C++, Fortran compiler — main differences in how and when it is invoked

---

**Block Diagram of a Compiler**

**Compiler Passes:**
- Scanner
- Parser
- Semantic Routines
- Optimizer
- Code Generator

**Scanner**
- Tokenizer, lexer, also processes comments and directives.
- Token description via regular expressions → scanner generators. Takes non-trivial time.

**Parser**
- Grouping of tokens. CFG (context free grammar). Error detection and recovery. Parser generator tools.

**Semantic Routines**
- The heart of a simple compiler. Deals with the meaning of the language constructs. Translation to intermediate representation (IR). Abstract code generation. Not automated, but can be formalized through Attribute Grammars.
- Generate functionally equivalent but improved code. Complex. Slow. User options to set level. Peephole vs. global optimization. Source vs. object code optimization. Usually hand-coded. Automation is a research topic, e.g. template optimizers.

**Optimizer**
- Generating AST supports other automatic techniques
- Optimization frameworks, and automatic generators are research topics
- Table & template driven code generators (i.e. BURS) are used in, e.g. GCC and Jikes RVM.

---

**Job Description of a Compiler**

- At a very high level a compiler performs two steps:
  - Analyze the source program
  - Synthesize (generate) the target code

---

**Compiler Tools**

**Compiler Passes:**
- Bulk of the work is still manual
- Lex generates scanners from high level input
- Yacc generates parsers from high level input. Symbol table routines are available
- Generating AST supports other automatic techniques
- Optimization frameworks, and automatic generators are research topics
- Table & template driven code generators (i.e. BURS) are used in, e.g. GCC and Jikes RVM.
**INPUT, OUTPUT AND INTERMEDIATE REPRESENTATIONS**

```
character sequence  \( I F (a < b) \text{ THEN } c = d + e \)
```

```
Scanner

IF

Parser

cond_expr

```

```
syntax tree

\[ \text{IF, stmt, cond_expr, \langle a < b \rangle, then_clause, list, assgn_stmt, rhs, \langle c = d + e \rangle, \langle \text{GE, a, b, L1} \rangle, \langle \text{ADD, \langle d, e \rangle, c} \rangle, \langle \text{Label, L1} \rangle] \]

Semantic Routines

```

```
3-address code

Code Generator

```

```
assembly code

```

**SEQUENCE OF COMPILER PASSES**

In general, all compiler passes are run in sequence. When run:

- They read the internal program representation,
- process the information, and
- generate the output representation.

For a very simple compiler, simplifications can be made. For example:

- Semantic routines and code generator are combined
- There is no optimizer
- All passes may be combined into one. That is, the compiler performs all steps in one run.
- One-pass compilers do not need an internal representation. They process a syntactic unit at a time, performing all steps from scanning to code generation.
- This is simpler than our project compiler

Example: (simple) Pascal compilers

**LANGUAGE SYNTAX AND SEMANTICS**

- **Syntax** defines the structure of a language. E.g., an IF clause has the structure:

  ```
  I F \{ \text{ expression } \} \text{ THEN } \text{ statements}
  ```

- **Semantics** defines its meaning. E.g., an IF clause means:

  ```
  test the expression; if it evaluates to true, execute the statements.
  ```

**SYNTACTIC & SEMANTIC PHASES**

```
character sequence  \( I F (a < b) \text{ THEN } c = d + e \)
```

```
Scanner

IF

Parser

cond_expr

```

```
syntax tree

\[ \text{IF, stmt, cond_expr, \langle a < b \rangle, then_clause, list, assgn_stmt, rhs, \langle c = d + e \rangle, \langle \text{GE, a, b, L1} \rangle, \langle \text{ADD, \langle d, e \rangle, c} \rangle, \langle \text{Label, L1} \rangle] \]

Semantic Routines

```

```
3-address code

Code Generator

```

```
assembly code

```

```
CONTEXT-FREE SYNTAX

- The context-free syntax part specifies legal sequences of symbols, independent of their type, scope and meaning.
  - $a=b+c$; is valid syntax even if $a$, $b$ and $c$ are not declared.
  - *Bob swam through the concrete* is grammatically valid English even though semantically it is nonsense.
- Called context-free because the context (i.e. what surrounds the symbols being examined) does not affect their processing. The processing is independent of the context.

CONTEXT-SENSITIVE SYNTAX

- The context-sensitive syntax part defines restrictions imposed by type and scope.
  - Also called the static semantics.
    - all identifiers must be declared
    - operands must be type compatible
    - correct number of parameters.
  - Can be specified informally or through attribute grammar.
  - Context sensitive because surrounding context (i.e. identifier types, number of parameters, …) affects the interpretation.

SYMBOL AND ATTRIBUTE TABLES

- Key repository of context sensitive symbol information
- Keep information about identifiers: variables, procedures, labels, etc.
- The symbol table is used by most compiler passes
  - Symbol information is entered at declaration points.
  - Checked and/or updated where the identifiers are used in the source code and as a result of program analysis

EXAMPLE - CONTEXT-FREE AND CONTEXT-SENSITIVE SYNTAX PARTS

- CFG:
  $$ E1 \rightarrow E2 + T $$
  “The term $E1$ is composed of an $E2$, a ‘$+$’, and a $T$”

- Attribute Grammar:
  $$ (E2.type=numeric) \text{ and } (T.type=numeric) $$
  “Both $E1$ and $T$ must be of type numeric”
EXECUTION SEMANTICS

(a.k.a. *runtime semantics*)

- Often specified informally
  - Java virtual machine semantics, C & Fortran machine models
  - "Verification" by testing, compliance kits

- Attempts to formalize execution semantics:
  - Operational or interpreter model: (state-transition model). E.g., *Vienna definition language*, used for PL/1. Large, verbose, reads like a contract.
  - *Axiomatic definitions*: specifies the effect of statements on variable relationships. More abstract than operational model.
  - Hope of more verification done automatically, but progress is slow

SIGNIFICANCE OF SEMANTIC SPECIFICATION

- Leads to a well-defined language, that is complete and unambiguous.
- Automatic generation of semantics routines becomes possible.
- Note: compiler is a *de facto* language definition. (what's not fully defined in the language specs is defined in the compiler) It is not necessarily a *good* language definition!
  - Before ANSI and ISO C, K&R (AT&T) C compiler was the *de facto* C standard

COMPILER AND LANGUAGE DESIGN

- Language design and the capabilities of compilers are strongly inter-related
- Architectural features and compiler design are strongly inter-related

There is a strong mutual influence:
- *hard* to compile languages are often *hard* to read (e.g. C++)
- *easy* to compile languages lead to quality compilers, better code, smaller compiler, more reliable, cheaper, wider use, better diagnostics.

Example: *dynamic typing*
- *seems* convenient because type declaration is not needed
- However, such languages can be
  + *hard* to read because the type of an identifier is not known
  + *hard* to compile into efficient binary because the compiler cannot make assumptions about the identifier’s type.
CISC: complex instructions were available at the assembly language level, e.g. instructions to perform a procedure call, evaluate a polynomial, do text replacement, etc.

- Complex instructions often implemented in microcode.
- Complex instructions often slower than a sequence of faster instructions.
- Even if complex instruction faster, must figure out when to use it. To generate an evaluate polynomial instruction, a compiler must recognize this operation in a program – not easy.

RISC design principles came out of IBM Yorktown 801 project and Stanford RISC project, which in turn came out of compiler work targeting subset of very CISCy System 360 instruction set.

- CISC motivated in large part by high level instructions to make assembly programming easier, and to tune hardware to high level ops.
- RISC motivated in large part to make hardware "simpler" and amenable to compilers.

So Far We Have Covered ...

Task of compilers, interpreters, assemblers
- Compiler passes and intermediate representations
- Scope of compiler writing tools
- Terminology: Syntax, semantics, context-free grammar, context-sensitive parts, static semantics, runtime/execution semantics
- Specification methods for language semantics
- Compiler, language and architecture design

Next: An example compiler
IMPLEMENTING A MICRO COMPILER

- 1-pass compiler. No explicit intermediate representation.
- Scanner: tokenizes input character stream. Is called by parser on-demand.
- Parser recognizes syntactic structure, calls Semantic Routines.
- Semantic routines, in turn, call code generation routines directly, producing code for a 3-address virtual machine.
- Symbol table is used by Semantic routines only.

THE MICRO LANGUAGE

- integer data type only
- implicit identifier declaration. 32 chars max. [A-Z][A-Z0-9]*
- literals (numbers): [0-9]*
- comment: -- non-program text <end-of-line>
- Program:
  BEGIN Statement, Statement, ... END

MICRO LANGUAGE

- Statement:
  + Assignment:
    ID := Expression
    Expression can contain infix + -, ( ), Ids, Literals
    Note: no unary minus (i.e. 0-27 ok, -27 not.)
  + Input/Output:
    READ(ID, ID, ...)
    WRITE(Expression, Expression, ...)

MICRO SCANNER (LEXICAL ANALYZER)

Interface used by parser: token scanner();

typedef enum token_types {
  Begin, End, Read, Write, ID, Intliteral, Lparen, Rparen, Semicolon, Comma, Assignop, Plusop, Minusop, ScanEof} token;

Scanner Algorithm: (see textbook p. 28/29)
**Scanner Operation**

- **Scanner routine:**
  - What the scanner can identify corresponds to what a regular expression (and its corresponding finite state automata) can recognize.
  - Identifies the next token in the input character stream:
    - Read a token
    - Identify its type
    - Return token type and "value"

**Recognizing Tokens**

- Skip spaces.
- If the first non-space character is a:
  - letter: read until non-alphanumeric. Put in buffer. Check for reserved words. Return reserved word or identifier.
  - (next character an =) → return ASSIGNOP token.
  - (next character also a -) → a comment.
  - Skip to EOL.
  - Read another token.
  - Otherwise return MINUSOP.
- "unget" the next character that had to be read for Ids, reserved words, numbers, and minusop.

*Note: Read-ahead by one character is necessary.*

**Grammar and Parsers**

- Context-Free Grammar (CFG) is most often used to specify language syntax.
- (Extended) Backus-Naur Form (BNF) is a convenient notation. Named after John Backus (Fortran) and Peter Naur (Algol 60)
- It includes a set of rewriting rules or Productions.
  - A production tells us how to compose a non-terminal from terminals and other non-terminals.

**Micro Grammar (Fig. 2.4)**

- Program ::= BEGIN Statement-list END
- Statement-list ::= Statement [Statement]
- Statement ::= ID ::= Expression ; |
  - READ (Id-list) ; |
  - WRITE (Expr-list) ; |
- Id-list ::= ID , ID |
- Expr-list ::= Expression , Expression |
- Expression ::= Primary (Add-op Primary) |
- Primary ::= (Expression) |
  - ID |
  - INFLITERAL |
- Add-op ::= PLUSOP , MINUSOP |
- System-goal ::= Program SCANEOF

Words in **bold** are terminals, in non-bold are non-terminals.
HOW ARE GRAMMARS AND PROGRAMS RELATED?

- Consider the Micro program

```
BEGIN id := id + id; END
```

- This program can be generated by the grammar by rewriting non-terminals

- We start with the goal production non-terminal (i.e. `program`)

---

GRAMMARS AND PROGRAMS

- Program ::= BEGIN Statement-list END

- Rewriting `Program` with the right hand side gives:

```
BEGIN Stmt-list END
```

Next rewrite the non-terminal `Stmt-list` using the rule

- Statement-list ::= Statement {Statement}

where `{…}` denotes zero or more repetitions of the enclosed non-terminal. Rewriting `Statement-list` gives

```
BEGIN Statement {Statement} END
```

Continuing with where we left off: `BEGIN Statement {Statement} END`

Rewrite the non-terminal `Statement` using the production

```
Statement ::= ID := Expression ;
```

Producing

```
BEGIN ID := Expression ; {Statement} END
```

Rewrite the non-terminal `Expression` with

```
Expression ::= Primary { Add-op Primary }
```

producing

```
BEGIN ID := Primary { Add-op Primary } ; {Statement} END
```

Rewrite the first non-terminal `Primary` in

```
BEGIN ID := Primary { Add-op Primary } ; {Statement} END
```

using the production

```
Primary ::= ID
```

Producing:

```
BEGIN ID := ID { PLUSOP Primary} ; {Statement} END
```

Rewrite the non-terminal for `Add-op` using the rule

```
Add-op ::= PLUSOP
```

Produces:

```
BEGIN ID := ID { PLUSOP Primary} ; {Statement} END
```

Rewrite the non-terminal `Primary` using the rule

```
Primary ::= ID
```

```
BEGIN ID := ID { PLUSOP ID} ; {Statement} END
```
The meta-characters '{' and '}' can go away, and we are not going to use the optional Statement, so they go away.

Moreover, PLUSOP is the name of the + token, and so:

BEGIN ID := ID PLUSOP ID : END

The goal of parsing is to do the inverse: read in a string of tokens and matches them with the productions that generated the string of tokens.

As each production is recognized, actions can be taken to represent the semantics of the recognized production.

The overall operation of Micro's parser:

• start at goal term, rewrite productions (from left to right)
  + when it is a terminal: check if it matches an input token
  + otherwise (it is a non-terminal, must parse another production):
    • when there is a single choice for a production use that production, (e.g. see the start production, id-list, etc.)
    • else: take the production that matches the first token (e.g. when parsing statement the kind of statement (and corresponding production) is indicated by READ, WRITE, ID.

Lack of expected token means a syntax error

Notes:
• 1-token lookahead is necessary (to match 1st token).
• In Micro, Static (CS) semantics are not checked.

Each production P has an associated procedure, usually named after the nonterminal on the left hand side (LHS).

Consider the grammar at the right for some language, where λ is the empty or null terminal symbol.

Generated strings:
"tu": X ⇒ AB ⇒ tB ⇒ tu
"tv": X ⇒ AB ⇒ tB ⇒ tv
"t": X ⇒ AB ⇒ tB ⇒ t.

Algorithm for creating routine P( ) to parse productions with left hand sides P:
• for nonterminal A on the right-hand side (RHS) : call A( ).
• for terminal t on the RHS : call match(t), (matching the token t from the scanner).
• if there is a choice for a production B: look at First(B), where First(B) is the set of terminals that B can start with
  • For B in the grammar above, First(B) = {u, v} in the above grammar.
  • First(B) distinguishes all choices among non-empty productions in an LL(1) grammar.
  • Empty productions are used only if there is no other choice.
  • First(B) defines the branches of a case or if in the parse routine for B()
A RECURSIVE DESCENT PROCEDURE

Program \[\Rightarrow \text{BEGIN} \text{ Statement-list END}\]

Procedure Program( ) {
  match(BEGIN);
  StatementList( );
  match(END);
}

PARSER CODE FOR MICRO

(text pages 36 – 38) Things to note:

+ there is one procedure for each non-terminal.
+ nonterminals with choices (e.g. Statement) have case or if statements.
+ an optional list is parsed with a loop construct, testing the First() set of the list item.
+ error handling is minimal.

ANOTHER PARSER PROCEDURE

id-list \[\Rightarrow \text{ID} \ (,\text{ID})\]

Procedure IdList( ) {
  match(ID);
  WHILE LookAhead(COMMA) {
    match(COMMA);
    match(ID);
  }
}

SEMANTIC PROCESSING AND CODE GENERATION

× Micro will generate code for a 3-address machine:

| OP A,B,C   | performs A op B \rightarrow C |

× Temporary variables may be needed to convert expressions into 3-address form. Naming scheme: Temp&1, Temp&2, ...

\[D=\text{A+B}\times \text{C} \quad \text{MULT} \ B,C,\text{TEMP}\&1\]
\[\text{ADD} \ A,\text{Temp}\&1,\&\text{Temp2}\]
\[\text{STORE} \ &\text{Temp2},D\]
How can we facilitate the creation of the semantic routines?

Idea: call routines that generate 3-address code at the right points during parsing.

These action routines will do one of two things:
1. Collect information about parsed symbols for use by other action routines. The information is stored in semantic records.
2. Generate the code using information from semantic records and the current parse procedure.

Note interaction with precedence of operators!

Annotations are inserted in the grammar, specifying when semantics routines are to be called.

Consider A=B+2:
+ num( ) and id( ) write semantic records containing ID names and number values.
+ addop( ) generates code for the expr production, using information from the semantic records created by num( ) and id( ). A temporary variable is created.
+ assign( ) generates code for the assignment to A, using the result of B+2 generated by addop( )

Annotations in Chap. 2 print information about what the parser has recognized.

At #start, nothing has been recognized, so this takes no action. End of parse is recognized by the final production:

System-goal ::= Program SCANEOF #finish

In a production compiler, the #start routine might set up program initialization code (i.e., initialization of heap storage and static storage, initialization of static values, etc.) and #finish would clean up the parser-only data structures.
No semantic actions are associated with this statement because the necessary semantic actions associated with statements are done when a statement is recognized.

Different semantic actions used when the parser finds an expression. In Expr-list, it is handled with write_expr; in Primary we choose to do nothing, but could have expressed a different semantic action if there were a reason to do so.

We know that different productions, or rules of the grammar, are reached in different ways, and can tailor semantic actions (and the grammar) appropriately.

• Note that in the grammar of Fig. 2.4, there is no Ident nonterminal.
  • An Ident a placeholder is created to allow semantic actions as the nonterminal is processed.
  • The programs look syntactically the same, but the additional productions allow the semantics to be richer.

Semantic actions create a semantic record for the ID and thereby create a way to pass the id being read to the read_id semantic routine.

A procedure corresponds to each annotation of the grammar.

The parsing routines have been extended to return information about the identified constructs. E.g.,

```c
void expression(expr_rec *results)
```
SO FAR WE HAVE COVERED

- Lecture 1: structure of compilers and terminology
- Lecture 2: scanner, parser, semantic routines and code generation for a one-pass compiler for the Micro language

Next: Scanning / Lexical analysis

OPERATOR PRECEDENCE

- Operator precedence is also specified in the
- CFG tells both what is legal syntax and order it is parsed.

For example,

Expr ::= Factor { + Factor }
Factor ::= Primary { * Primary }
Primary ::= ( Expr ) | ID | INTLITERAL

Must finish “*” production before “+” production specifies the usual precedence rules: * before +