COM SCI 132 Week 1

Aidan Jan

April 4, 2024

Compilers vs Interpreters

- Compilers compile all the code into an executable file, and later, the OS can run the program straight through
- Interpreters read and execute code at the same time.

Compilers

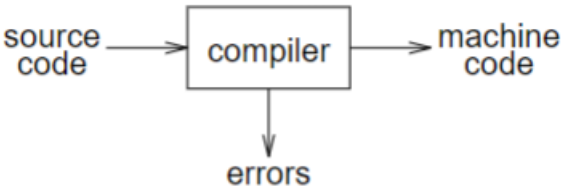
Qualities

What are some important qualities of a compiler?

1. Correct code
2. Output runs fast
3. Compiler runs fast
4. Compile time proportional to program size
5. Support for separate compilation
6. Good diagnostics for syntax errors
7. Works well with the debugger
8. Good diagnostics for flow anomalies
9. Cross language calls
10. Consistent, predictable optimzation

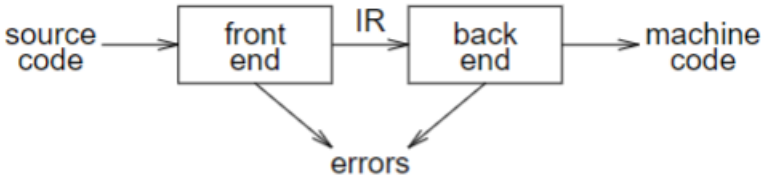
Abstract view

- Implications:
  - recognize legal (and illegal) programs
  - generate correct code
  - manage storage of all variables and code
  - agreement on format for object (or assembly) code
  - Big step up from assembler to higher level notations



Two-pass compilers

- Two pass compilers feature an *intermediate representation* (IR).



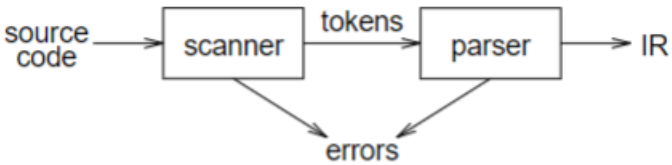
- Implications
  - Intermediate representation (IR)
  - Front end maps legal code into IR
  - Back end maps IR onto target machine
  - Simplify retargeting
  - Allows multiple front ends
  - Multiple passes ⇒ better code

Front end

Responsibilities:

- Recognize legal procedure
- Report errors
- Produce IR
- Preliminary storage map
- Shape the code for the back end

Much of front end construction can be automated.



Scanner:

- Maps characters into *tokens* - the basic unit of syntax

```
x = x + y;
```

becomes

```
<id, x> = <id, x> + <id, y>;
```

- character string value for a *token* is a *lexeme*
- typical tokens: *number*, *id*, +, -, *, /, `do`, `end`
- eliminates white space (tabs, blanks, comments)
- a key issue is speed ⇒ use specialized recognizer (as opposed to `lex`)

Parser:

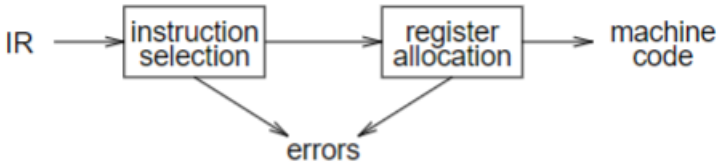
- recognize context-free syntax
- guide context-sensitive analysis
- construct IR(s)
- produce meaningful error messages
- attempt error correction

Parser generators mechanize much of the work.

Back end

- translate IR into target machine code
- choose instructions for each IR operation
- decide what to keep in registers at each point
- ensure conformance with system interfaces

Automation has been less successful here.



Instruction Selection:

- produce compact, fast code
- use available addressing modes
- pattern matching problem
  - *ad hoc* techniques
  - tree pattern matching
  - string pattern matching
  - dynamic programming

Register Allocation:

- have value in a register when used

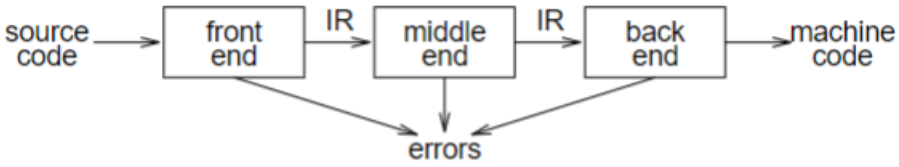
- limited resources
- changes instruction choices
- can move loads and stores
- optimal allocation is difficult

Modern allocators often use an analogy to graph coloring

Optimizing compiler

Code improvement:

- analyzes and changes IR
- goal is to reduce runtime
- must preserve values

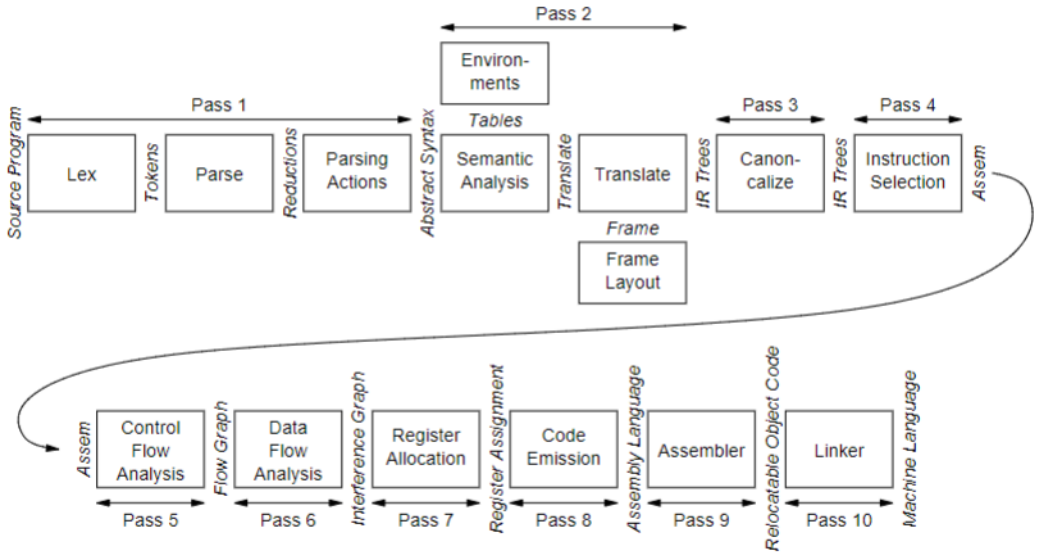


Optimizer (middle end)

Modern optimizers are usually built as a set of passes. Typical passes:

- constant propagation and folding
- code motion
- reduction of operator strength
- common subexpression elimination
- redundant store elimination
- dead code elimination

Compiler Example and Phases



Lex	Break source file into individual words, or tokens
Parse	Analyse the phrase structure of program
Parsing Actions	Build a piece of *abstract syntax tree* for each phrase
Semantic Analysis	Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase
Frame Layout	Place variables, function parameters, etc., into activation records (stack frames) in a machine-dependent way
Translate	Produce *intermediate representation trees* (IR trees), a notation that is not tied to any particular source language or target machine
Canonicalize	Hoist side effects out of expressions, and clean up conditional branches, for convenience of later phases
Instruction Selection	Group IR-tree nodes into clumps that correspond to actions of target-machine instructions
Control Flow Analysis	Analyse sequence of instructions into *control flow graph* showing all possible flows of control program might follow when it runs
Data Flow Analysis	Gather information about flow of data through variables of program; e.g., *liveness analysis* calculates places where each variable holds a still-needed (*live*) value
Register Allocation	Choose registers for variables and temporary values; variables not simultaneously live can share same register
Code Emission	Replace temporary names in each machine instruction with registers

Lexical Analysis

Patterns for the Scanner

Recall the scanner in the front end compiler.
A scanner must recognize various parts of the language's syntax For example, the easy parts: *white space*:

```
<ws> ::= <ws> ' '
       | <ws> '\t'
       | ' '
       | '\t'
```

keywords and operators

`do, end, etc.`

comments:

`//, /* ... */`

Other parts are much harder:
identifiers

- alphabetic followed by k alphanumerics (_, $, &, ...)

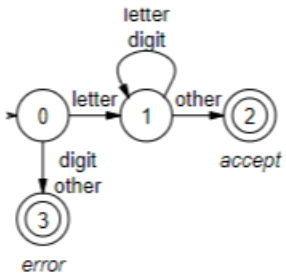
numbers

- integers: 0 or digit from 1-9 followed by digits from 0-9
- decimals: integer '.' digits from 0-9
- reals: (integer or decimal) 'E' (+ or -) digits from 0-9
- complex: '(' real ',' real ')'

These can be done easily with REGEX.

Recognizers

From a regular expression we can construct a *deterministic finite automaton* (DFA).



- letter $\rightarrow (a|b|c|\ldots|z|A|B|\ldots|Z)$
- digit $\rightarrow (0|1|2|\ldots|9)$
- id $\rightarrow letter(letter|digit)^*$

Tables for the Recognizer

`char_class:`

	$a-z$	$A-Z$	$0-9$	other
value	letter	letter	digit	other

`next_state:`

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

To change languages, we can just change tables.

Automatic Construction

Scanner generators automatically construct code from regular-expression-like descriptions

- construct a *dfa*
- use state minimization techniques
- emit code for the scanner (table driven or direct code)

A key issue in automation is an interface to the parser
For example, `lex` is a scanner generator supplied with UNIX.

- emits C code for scanner
- provides macro definitions for each token (used in the parser)

LL Parsing

Recall, the role of the Parser, the second part of the front-end compiler.

Syntax Analysis

Context-free syntax is specified with a context-free grammar.
Formally, a CFG G is a 4-tuple (V_t, V_n, S, P), where

- V_t is the set of *terminal* symbols in the grammar. For our purposes, V_t is the set of tokens returned by the scanner.
- V_n, the *nonterminals*, is the set of syntactic variables that denote sets of (sub)strings occurring in the language. These are used to impose a structure on the grammar.
- S is the distinguished nonterminal ($S \in V_n$) denoting the entire set of strings in $L(G)$. This is sometimes called a *goal symbol*.
- P is a finite set of *productions* specifying how terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its left hand side.

The set $V = V_t \cup V_n$ is called the *vocabulary* of G.

Grammars are often written in Backus-Naur form (BNF). For example:

```
<goal> ::= <expr>
<expr> ::= <expr><op><expr>
         |  num
         |  id
<op>   ::= +
         |  -
         |  *
         |  /
```

This describes simple expressions over numbers and identifiers. In a BNF for a grammar, we represent

1. non-temrinals with angle brackets or capital letters
2. terminals with `typewriter` font or underline
3. productions as in the example

Scanning vs. Parsing - where do we draw the line?

```
term ::= [a-zA-Z]([a-zA-Z=]|[0-9])*
      |  0|[1-9][0-9]*
op   ::= +|-|*|/
expr ::= (term op)*term
```

Regular expressions are used to classify:

- identifiers, numbers, keywords
- REs are more concise and simpler for tokens than a grammar
- more efficient scanners can be built from REs (DFAs) than grammars

Context-free grammars are used to count:

- brackets: `(), begin...end, if...then...else`
- imparting structure: expressions

Syntactic analysis is complicated enough: grammar for C has around 200 productions. Factoring out lexical analysis as a separate phase makes the compiler more manageable.

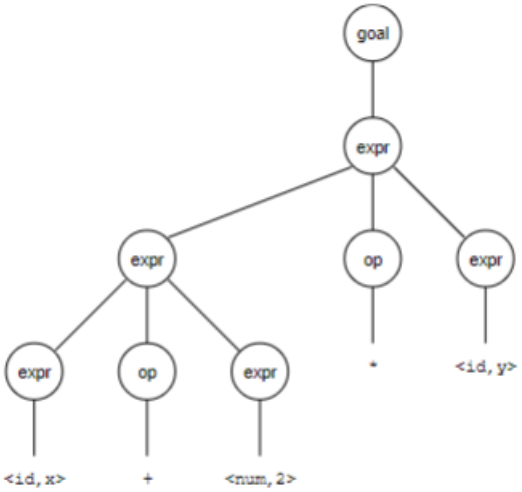
Derivations

We can view the productions of a CFG as rewriting rules. Using our exmple CFG:

```
<goal> => <expr>
       => <expr><op><expr>
       => <expr><op><expr><op><expr>
       => <id,x><op><expr><op><expr>
       => <id,x> + <expr><op><expr>
       => <id,x> + <num,2><op><expr>
       => <id,x> + <num,2> * <expr>
       => <id,x> + <num,2> * <id,y>
```

- We have derived the sentence `x + 2 * y`.
- We denote this `<goal>=>* id + num * id`.
- Such a sequence of rewrites is a *derivation* or a *parse*.
- The process of discovering a derivation is called *parsing*.
- At each step, we chose a non-terminal to replace.
- This choice can lead to different derivations.
- Two are of particular interest:
  - leftmost derivation: the leftmost non-terminal is replaced at each step
  - rightmost derivation: the rightmost non-terminal is replaced at each step
- *The previous example was a leftmost derivation.*

Precedence



A tree can be generated using the CFG. However, for the tree shown here, the *treewalk evaluation* computes `(x + 2) * y` - the wrong answer!
The two derivations point out a problem with the grammar: it has no notion of precedence, or *implied order* or evaluation. To add precedence takes additional machinery:

```
<goal>   ::= <expr>
<expr>   ::= <expr> + <term>
           |  <expr> - <term>
           |  <term>
<term>   ::= <term> * <factor>
           |  <term> / <factor>
           |  <factor>
<factor> ::= num
           |  id
```

This grammar enforces a precedence on the derivation:

- terms *must* be derived from expressions
- forces the "correct" tree

Ambiguity

If a grammar has more than one derivation for a single sentential form, then it is **ambiguous**.
Example:

```
<stmt> ::= if <expr> then <stmt>
         |  if <expr> then <stmt> else <stmt>
         |  other stmts
```

Consider deriving the sentential form:

$$\texttt{if } E_1 \texttt{ then if } E_2 \texttt{ then } S_1 \texttt{ else } S_2$$

This has two derivations, which leads to ambiguity that is purely grammatical.
It is a *context-free* ambiguity.

Eliminating Ambiguity

It may be possible to eliminate ambiguities by rearranging the grammar:

```
<stmt>      ::= <matched>
             |  <unmatched>
<matched>   ::= if <expr> then <matched> else <matched>
             |  other stmts
<unmatched> ::= if <expr> then <stmt>
             |  if <expr> then <matched> else <unmatched>
```

This generates the same language as the ambiguous grammar, but applies the common sense rule:

match each `else` with the closest unmatched `then`

This is most likely the language designer's intent.

Overloading

Ambiguity is often due to confusion in the context-free specification. Context-sensitive confusions can arise from *overloading*. For example:

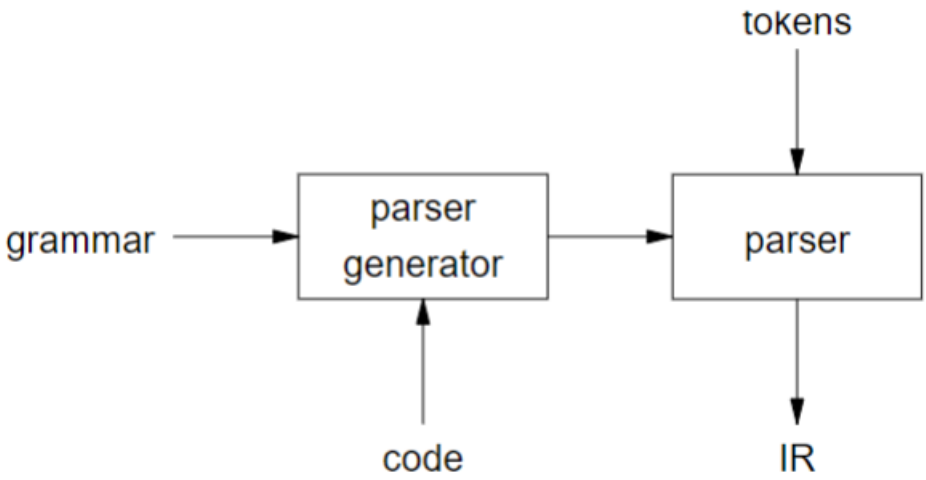
```
a = f(17)
```

In many Algol-like languages, `f` could be a function or subscripted variable. Disambiguating this statement requires context:

- need *values* of declarations
- not *context-free*
- really an issue of *type*

Rather than complicate parsing, we will handle this separately.

Parsing: the big picture



Our goal is a flexible parser generator system.

Top-Down vs. Bottom-Up Parsing

Top-down parsers:

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- may require backtracking
- some grammars are backtrack-free (predictive)

Bottom-up parsers:

- start at the leaves and fill in
- start in a state valid for legal first tokens
- as input is consumed, change state to encode possibilities (recognize valid prefixes)
- use a stack to store both state and sentential forms

Top-down Parsing

A top-down parser starts with the root of the parse tree, labelled with the start or goal symbol of the grammar. To build a parse, it repeats the following steps until the fringe of the parse tree matches the input string

1. At a node labelled A, select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α
2. When a terminal is added to the fringe that doesn't match the input string, backtrack.
3. Find the next node to be expanded (must have a label in V_n)

The key in selecting the right production in step 1 $\Rightarrow$ should be guided by input string
Example using $x - 2 * y$:

Prod'n	Sentential form	Input
–	⟨goal⟩	↑x – 2 * y
1	⟨expr⟩	↑x – 2 * y
2	⟨expr⟩ + ⟨term⟩	↑x – 2 * y
4	⟨term⟩ + ⟨term⟩	↑x – 2 * y
7	⟨factor⟩ + ⟨term⟩	↑x – 2 * y
9	id + ⟨term⟩	↑x – 2 * y
–	id + ⟨term⟩	x ↑– 2 * y
–	⟨expr⟩	↑x – 2 * y
3	⟨expr⟩ – ⟨term⟩	↑x – 2 * y
4	⟨term⟩ – ⟨term⟩	↑x – 2 * y
7	⟨factor⟩ – ⟨term⟩	↑x – 2 * y
9	id – ⟨term⟩	↑x – 2 * y
–	id – ⟨term⟩	x ↑– 2 * y
–	id – ⟨term⟩	x – ↑2 * y
7	id – ⟨factor⟩	x – ↑2 * y
8	id – num	x – ↑2 * y
–	id – num	x – 2 ↑* y
–	id – ⟨term⟩	x – ↑2 * y
5	id – ⟨term⟩ * ⟨factor⟩	x – ↑2 * y
7	id – ⟨factor⟩ * ⟨factor⟩	x – ↑2 * y
8	id – num * ⟨factor⟩	x – ↑2 * y
–	id – num * ⟨factor⟩	x – 2 ↑* y
–	id – num * ⟨factor⟩	x – 2 * ↑y
9	id – num * id	x – 2 * ↑y
–	id – num * id	x – 2 * y ↑

Another possible parse for $x - 2 * y$

Prod'n	Sentential form	Input
–	⟨goal⟩	↑x − 2 ∗ y
1	⟨expr⟩	↑x − 2 ∗ y
2	⟨expr⟩ + ⟨term⟩	↑x − 2 ∗ y
2	⟨expr⟩ + ⟨term⟩ + ⟨term⟩	↑x − 2 ∗ y
2	⟨expr⟩ + ⟨term⟩ + ⋯	↑x − 2 ∗ y
2	⟨expr⟩ + ⟨term⟩ + ⋯	↑x − 2 ∗ y
2	⋯	↑x − 2 ∗ y

If the parser makes the worng choices, expansion doesn't terminate. This isn't a good property for a parser to have. (Parsers should terminate!)

Left-recursion

Top-down parsers cannot handle left-recursion in a grammar. Formally, a grammar is *left-recursive* if:

$$\exists A \in V_n \text{ such that } A \Rightarrow^+ A\alpha \text{ for some string } \alpha$$

Our simple expression grammar is left-recursive.

Eliminating left-recursion

To remove left-recursion, we can transform the grammar. Consider the grammar fragment:

```
<foo> ::= <foo> α
       |  β
```

where α and β do not start with <foo>
We can rewrite this as:

```
<foo> ::= β <bar>
<bar> ::= α <bar>
       | ε
```

where <bar> is a new non-terminal. This fragment contains no left-recursion.
Example: Our expression grammar contaions two cases of left-recursion.

```
<expr>   ::= <expr> + <term>
          |  <expr> - <term>
          |  <term>
<term>   ::= <term> * <factor>
          |  <term> / <factor>
          |  <factor>
```

Applying the transformation gives:

```
<expr>   ::= <term><expr'>
<expr'>  ::= + <term><expr'>
          |  - <term><expr'>
          |  ε
<term>   ::= <factor><term'>
<term'>  ::= * <factor><term'>
          |  / <factor><term'>
          |  ε
```

With this grammar, a top-down parser will:

- terminate
- backtrack on some inputs

Although it is longer and harder to read, it factors out left-recursion.

How much lookahead is needed?

We saw that top-down parsers may need to backtrack when they select the wrong production.
Do we need arbitrary lookahead to parse CFGs?

- in general, yes
- use the Earley or Cocke-Younger, Kasami algorithms

Fortunately,

- large subclasses of CFGs can be parsed with limited lookahead
- most programming language constructs can be expressed in a grammar that falls in these subclasses

Among the interesting subclasses are:

- LL(1): left to right scan, **left**-most derivation, 1-token lookahead
- LR(1): left to right scan, **right**-most derivation, 1-token lookahead

Predictive Parsing

Basic idea: For any two productions $A \rightarrow \alpha|\beta$, we would like a distinct way of choosing the correct produciton to expand.
For some RHS $\alpha \in G$, define FIRST(α) as the set of tokens that appear first in some string derived form α. That is, for some $w \in V_t^*$, $w \in$ FIRST(α) if and only if $\alpha \Rightarrow^* w\gamma$.
Key property: Whenever two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \phi$$

This would allow the parser to make a correct choice with a lookahead of only one symbol! The example grammar has this property!

If a grammar does not have this property, we can transform it to have the property:

- For each non-terminal A find the longest prefix α common to two or more of its alternatives.
- if $\alpha \neq \epsilon$ then replace all of the A productions $A \rightarrow \alpha\beta_1|\alpha\beta_2|\cdots|\alpha\beta_n$ with:

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1|\beta_2|\cdots|\beta_n$$

  - where A' is a new non-terminal

- Repeat until no two altrernatives for a single non-terminal have a common prefix.