Syntax and Semantics of Prog. Languages

Syntax:
Describes what a legal program looks like

Semantics:
Describes what a correct (legal) program means

A formal language is a (possibly infinite) set of sentences (finite sequences of symbols) over a finite alphabet $\Sigma$ of (terminal) symbols: $L \subseteq \Sigma^*$

Examples:

- $L = \{ \text{identifiers of length 2} \}$ with $\Sigma = \{a, b, c\}$
- $L = \{ \text{strings of only 1s or only 0s} \}$
- $L = \{ \text{strings starting with $\$\$ and ending with $\#\$}, \text{ and any combination of 0s and 1s inbetween } \}$
- $L = \{ \text{all syntactically correct Java programs} \}$
Syntax and Semantics: How does it work?

Syntactic representation of “values”

What do the following syntactic expressions have in common?

XI
1011
B
$\#$
$3 + 20 - (2 \times 6)$
Syntax and Semantics: How does it work?

Syntactic representation of “values”

What do the following syntactic expressions have in common?

XI
1011
B
$\|\|\|\|\|\|\|\|\|\|\#
3 + 20 − (2 \times 6)

Answer: They are possible representations of the integer value “11” (written as a decimal number)

What is computation?

Possible answer: A (finite) sequence of syntactic manipulations of value representations ending in a “normal form” which is called the result. Normal forms cannot be manipulated any further.
Syntax and Semantics: How does it work?

Here is a “game” (rewrite system):

**input**: Sequence of characters starting with \$ and ending with \#, and any combination of 0s and 1s inbetween.

**rules**: You may replace a character pattern \(X\) at any position within the character sequence on the left-hand-side by the pattern \(Y\) on the right-hand-side: \(X \Rightarrow Y\):

- **rule 1**: \(\$ 1 \Rightarrow 1 \&\)
- **rule 2**: \(\$ 0 \Rightarrow 0 \$\)
- **rule 3**: \(\& 1 \Rightarrow 1 \$\)
- **rule 4**: \(\& 0 \Rightarrow 0 \&\)
- **rule 5**: \(\$ \# \Rightarrow \rightarrow \text{A}\)
- **rule 6**: \(\& \# \Rightarrow \rightarrow \text{B}\)

Replace patterns using the rules as often as you can. When you cannot replace a pattern any more, stop.
Syntax and Semantics: How does it work?

example input:

$ 0 0 \#$

$\boxed{0} 0 \#$ is rewritten as $0 \boxed{\$} 0 \#$ by rule 2

$0 \boxed{\$} 0 \#$ is rewritten as $0 0 \boxed{\$} \#$ by rule 2

$0 0 \boxed{\$} \#$ is rewritten as $0 0 \rightarrow A$ by rule 6

no more rules can be applied (STOP)

More examples:

$0 1 1 0 1 \#$

$1 0 1 0 0 \#$

$1 1 0 0 1 \#$

Questions

• Can we get different “results” for the same input string?

• Does all this have a meaning (semantics), or are we just pushing symbols?
Syntax without Semantics?

The green apple is colorless.

Syntactically right, but semantically incorrect!
Review: Compilers

Implications:

- recognize legal (and illegal) programs
- generate correct code
- manage storage of all variables and code
- need format for object (or assembly) code

*Big step up from assembler – higher level notations*
Traditional two pass compiler

**Pass:** reading and writing entire program

```
source code → front end → `il` → back end → machine code
errors
```

**Implications:**

- intermediate language (*il*)
- front end maps legal code into *il*
- back end maps *il* onto target machine
- simplify retargeting
- allows multiple front ends
- multiple passes ⇒ better code

*Front end is O(n)*

*Back end is NP-Complete*
Front end of a compiler

Parser: syntax & semantic analyzer, il code generator
(syntax-directed translator)

Front End Responsibilities:

- recognize legal programs
- report errors
- produce il
- preliminary storage map
- shape the code for the back end

Much of front end construction can be automated
The syntax of programming languages is often defined in two layers: tokens and sentences.

- tokens – basic units of the language
  Question: How to spell a token (word)?
  Answer: regular expressions

- sentences – legal combination of tokens in the language
  Question: How to build correct sentences with tokens?
  Answer: (context-free) grammars (CFG)

- E.g., Backus-Naur form (BNF) is a formalism used to express the syntax of programming languages.
Formalisms for Lexical and Syntactic Analysis

1. Lexical Analysis: Converts source code into sequence of tokens.

2. Syntax Analysis: Structures tokens into parse tree.

Two issues in Formal Languages:

- **Language Specification** → formalism to describe what a valid program (sentence) looks like.

- **Language Recognition** → formalism to describe a machine and an algorithm that can verify that a program is valid or not.

For (2), we use context-free grammars to specify programming languages. Note: recognition, i.e., parsing algorithms using PDAs (push-down automata) will be covered in CS415.

For (1), we use regular grammars/expressions for specification and finite (state) automata for recognition.
Lexical Analysis (Scott 2.1, 2.2)

character sequence

if \( \leq \) then \( := \) 1

\[ i | f | a | \leq | b | t | h | e | n | c | : = | 1 \]

\[ \text{scanner} \]

\[ \text{if} \Rightarrow \text{id} <a> \Rightarrow \text{<=} \Rightarrow \text{id} <b> \]

\[ \Rightarrow \text{then} \Rightarrow \text{id} <c> \Rightarrow := \Rightarrow \text{num} <1> \]

token sequence

Tokens (Terminal Symbols of CFG, Words of Lang.)

- Smallest “atomic” units of syntax
- Used to build all the other constructs
- Example, C:

  keywords: for if goto volatile ...
  = * / - < > = <= >= <>
  ( ) [ ] ; := . , ...
  number (Example: 3.14 28 ...)
  identifier (Example: b square addEntry ...)

CS 314  fall’16 lecture 2, page 12
Lexical Analysis (cont.)

Identifiers

- Names of variables, etc.
- Sequence of terminals of restricted form; Example, C: A31, but not 1A3
- Upper/lower case sensitive?

Keywords

- Special identifiers which represent tokens in the language
- May be reserved (reserved words) or not
  - E.g., C: “if” is reserved.
  - E.g., FORTRAN: “if” is not reserved.

Delimiters – When does character string for token end?

- Example: identifiers are longest possible character sequence that does not include a delimiter
- Few delimiters in Fortran (not even ‘␣’)
  - DO I = 1.5 same as DOI=1.5
- Most languages have more delimiters such as ‘␣’, new line, keywords, ...
### Regular Expressions

A syntax (notation) to specify regular languages.

<table>
<thead>
<tr>
<th>RE $r$</th>
<th>Language $L(r)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>${a}$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>${\varepsilon}$</td>
</tr>
<tr>
<td>$r</td>
<td>s$</td>
</tr>
<tr>
<td>$rs$</td>
<td>${rs \mid r \in L(r), s \in L(s)}$</td>
</tr>
<tr>
<td>$r^+$</td>
<td>$L(r) \cup L(rr) \cup L(rrr) \cup \ldots$ (any number of $r$’s concatenated)</td>
</tr>
<tr>
<td>$r^*$</td>
<td>${\varepsilon} \cup L(r) \cup L(rr) \cup L(rrr) \cup \ldots$</td>
</tr>
<tr>
<td>$(r^* = r^+</td>
<td>\varepsilon)$</td>
</tr>
<tr>
<td>$(s)$</td>
<td>$L(s)$</td>
</tr>
</tbody>
</table>
### Examples of Expressions

<table>
<thead>
<tr>
<th>RE</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>bc</td>
</tr>
<tr>
<td>(a</td>
<td>b)c</td>
</tr>
<tr>
<td>aε</td>
<td></td>
</tr>
<tr>
<td>a*</td>
<td>b</td>
</tr>
<tr>
<td>ab*</td>
<td></td>
</tr>
<tr>
<td>ab*</td>
<td>c+</td>
</tr>
<tr>
<td>(a</td>
<td>b)*</td>
</tr>
<tr>
<td>(0</td>
<td>1)*1</td>
</tr>
<tr>
<td>RE</td>
<td>Language</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>a</td>
<td>bc</td>
</tr>
<tr>
<td>(a</td>
<td>b)c</td>
</tr>
<tr>
<td>aε</td>
<td>{a}</td>
</tr>
<tr>
<td>a*</td>
<td>b</td>
</tr>
<tr>
<td>ab*</td>
<td>{a, ab, abb, abbb, abbbbb, \ldots}</td>
</tr>
<tr>
<td>ab*</td>
<td>c+</td>
</tr>
<tr>
<td>(a</td>
<td>b)*</td>
</tr>
<tr>
<td>(0</td>
<td>1)*1</td>
</tr>
</tbody>
</table>
Regular Expressions for Programming Languages

Let letter stand for A | B | C | ... | Z
Let digit stand for 0 | 1 | 2 | ... | 9

integer constant:

identifier:

real constant:
Recognizers for Regular Expressions

Example 1: integer constant
RE: $\text{digit}^+$
FSA:

![FSA diagram for integer constant]

Example 2: identifier
RE: $\text{letter} \ (\text{letter} \ | \ \text{digit})^*$
FSA:

![FSA diagram for identifier]

Example 3: Real constant
RE: $\text{digit}^*.\text{digit}^+$
FSA:

![FSA diagram for Real constant]
A Finite-State Automaton is a quadruple:
\(< S, s, F, T >\)

- \(S\) is a set of states, e.g., \(\{S_0, S_1, S_2, S_3\}\)
- \(s\) is the start state, e.g., \(S_0\)
- \(F\) is a set of final states, e.g., \(\{S_3\}\)
- \(T\) is a set of labeled transitions, of the form
  \((\text{state}, \text{input}) \mapsto \text{state}\)
  [i.e., \(S \times \Sigma \to S\)
Finite State Automata

Transitions can be represented using a transition table:

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S1</td>
<td>S2</td>
<td></td>
</tr>
<tr>
<td>S1</td>
<td>S3</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>-</td>
<td>S3</td>
<td></td>
</tr>
</tbody>
</table>

An FSA accepts or recognizes an input string iff there is some path from its start state to a final state such that the labels on the path are that string.

Lack of entry in the table (or no arc for a given character) indicates an error—reject.
Practical Recognizers

- recognizer should be a deterministic finite automaton (DFA)
- read until the end of a token
- report errors (error recovery?)

identifier

\[
\begin{align*}
\text{letter} & \rightarrow (a \mid b \mid c \mid \ldots \mid z \mid A \mid B \mid C \mid \ldots \mid Z) \\
\text{digit} & \rightarrow (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9) \\
\text{id} & \rightarrow \text{letter} \ (\text{letter} \mid \text{digit})^*
\end{align*}
\]

Recognizer for identifier: (transition diagram)

\[
\begin{array}{c}
S0 \\
S1 \\
S2 \\
S3
\end{array}
\begin{array}{c}
\text{letter} \\
\text{digit} \\
\text{letter} \\
\text{digit}
\end{array}
\begin{array}{c}
\text{other} \\
\text{other} \\
\text{other} \\
\text{error}
\end{array}
\begin{array}{c}
\text{letter} \\
\text{digit} \\
\text{digit} \\
\text{error}
\end{array}
\begin{array}{c}
\text{other} \\
\text{other} \\
\text{error}
\end{array}
\begin{array}{c}
\text{error} \\
\text{error}
\end{array}
\]

fall’16 lecture 2, page 21
Implementation: Tables for the recognizer

Two tables control the recognizer.

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

| next_state: |
| class | S0 | S1 | S2 | S3 |
| letter | S1 | S1 | — | — |
| digit | S3 | S1 | — | — |
| other | S3 | S2 | — | — |

To change languages, we can just change tables.
Implementation: Code for the recognizer

```c
char ← next_char();
state ← S0; /* code for S0 */
done ← false;
token_value ← "" /* empty string */
while( not done ) {
    class ← char_class[char];
    state ← next_state[class,state];
    switch(state) {
        case S1: /* building an id */
            token_value ← token_value + char;
            char ← next_char();
            if (char == EOF)
                done = true;
            break;
        case S2: /* error state */
        case S3: /* error */
            token_type = error;
            done = true;
            break;
    }
}
return token_type;
```
**Improved efficiency**

Table driven implementation is slow relative to direct code. Each state transition involves:

1. classifying the input character
2. finding the next state
3. an assignment to the state variable
4. a trip through the case statement logic
5. a branch (while loop)

We can do better by “encoding” the state table in the scanner code.

1. classify the input character
2. test character class locally
3. branch directly to next state

This takes many fewer instructions.
Implementation: Faster scanning

S0: char ← next_char();
    token_value ← "" /* empty string */
    class ← char_class[char];
    if (class != letter)
        goto S3;

S1: token_value ← token_value + char;
    char ← next_char();
    class ← char_class[char];
    if (class != other)
        goto S1;
    if (char == DELIMITER )
        token_type = identifier;
        return token_type;
    goto S2;

S2: 

S3: token_type ← error;
    return token_type;
Next Lecture

Things to do:

- read Scott, Chapters 2.3 - 2.5