Decoration of parse tree for \((1 + 3) * 2\)

Program Assignment 2
Due date: 10/20 12:30pm

- Bitwise Manipulation of Hexidecimal Numbers
- CFG

\[
E \rightarrow E \mid A \quad \text{bitwise OR}
\]
\[
E \rightarrow A
\]
\[
A \rightarrow A \uparrow B \quad \text{bitwise XOR}
\]
\[
A \rightarrow B
\]
\[
B \rightarrow B \& C \quad \text{bitwise AND}
\]
\[
B \rightarrow C
\]
\[
C \rightarrow < C \quad \text{bitwise shift left 1}
\]
\[
C \rightarrow > C \quad \text{bitwise shift right 1}
\]
\[
C \rightarrow ~ C \quad \text{bitwise NOT}
\]
\[
C \rightarrow ( E )
\]
\[
C \rightarrow \text{hex}
\]
**LL(1) Attribute Grammar**

\[
E \rightarrow A \text{ EE} \\
\text{EE.st} = A \text{.val} \quad \text{EE.val} = E \text{E.val} \\
\text{EE.} \rightarrow A \text{ EE} \\
\text{EE.st} = \text{EE.st} | A \text{.val} \\
\text{EE.val} = \text{EE.val} \\
E \rightarrow \text{EE.val = EE.st} \\
A \rightarrow B \text{ AA} \\
\text{AA.st} = B \text{.val} \quad \text{AA.val} = A \text{.val} \\
\text{AA.} \rightarrow B \text{ AA} \\
\text{AA.st} = \text{AA.st} | B \text{.val} \\
\text{AA.val} = \text{AA.val} \\
A \rightarrow r \\
\text{AA.val} = \text{AA.st} \\
B \rightarrow C \text{ BB} \\
\text{BB.st} = C \text{.val} \quad \text{BB.val} = B \text{B.val} \\
\text{BB.} \rightarrow C \text{ BB} \\
\text{BB.st} = \text{BB.st} & C \text{.val} \\
\text{BB.val} = \text{BB.val} \\
B \rightarrow r \\
\text{BB.val} = \text{BB.st} \\
C \rightarrow C_1 < C_2 \\
C_1 \text{.val} = C_1 \text{.val} \ll 1 \\
( "\ll" \text{ bitwise shift left one} ) \\
C \rightarrow C_1 > C_2 \\
C_1 \text{.val} = C_1 \text{.val} \gg 1 \\
( "\gg" \text{ bitwise shift right one} ) \\
C \rightarrow C_1 ^ C_2 \\
C_1 \text{.val} = \sim C_2 \text{.val} \\
( \text{ "\^" bitwise NOT } ) \\
C \rightarrow \{ E \} \\
C \text{.val} = E \text{.val} \\
C \rightarrow \text{hex} \\
C \text{.val} = \text{hex.val} \\
\]

**Program Requirement**

- Write a C program using recursive descent parser w/ lexical analyzer to implement the designated inherited and synthesized attributes. The program evaluates the expressions in a file input.txt, and outputs the results to console.
- E.g., input: f&a 
  output: f&a = a
Program Requirements

• You cannot use more than 2 global/non-local variables, and they should be to hold the Operator and HexNumber as detected by the lexical analyzer.

Hints

• To solve the problems, you should take the following steps:
  – Write a lexical analyzer
  – Write a recursive-descent parser
  – Attributes are processed as either pass-in parameters or return value of functions
Hints

• Write a lexical analyzer
  – You may need to define an enum type for all possible tokens your scanner can generate
  – E.g., when reading hexadecimial numbers 0-9 or a-f, the recognized token is HEX, and the value is saved in HexNumber

Hints

• Write a recursive-descent parser
  – Parse the program by defining and invoking functions
  – E.g., $E \rightarrow A \ EE$
    $EE.st = A.val \ E.val = EE.val$
    ```c
    int E() {
        int val = A();
        return EE(val);
    }
    ```
Hints

• There are parameters passed in or returned when invoking functions. When invoking a function, the synthesized attribute is the return value, while the inherited attribute is the passing-in parameter.

Hints

• Sample code of main()

```c
int main() {
    int val;
    symbol = getNextToken();
    while (symbol != EOF_) {
        if (symbol != NEW_LINE) {
            val = E();
            printf(" = %x\n", val & 0xf);
        } else if (symbol == EOF_) break;
        symbol = getNextToken();
    }
    return 1;
}
```
Submission Requirements

• Pack the following files into a .tar file:
  – Source file: parser.c
  – Executable file: parser
  – Input file: input.txt
  – Output file: output.txt (copy all your console outputs to this file)
  – README file (optional, used if you have any additional comments/explanations about the files)
Dynamic Semantics

• Describe the meaning of expressions, statements, and program units
• No single widely acceptable notation or formalism for describing semantics
• Two common approaches:
  – Operational
  – Denotational

Operational Semantics

• *Gives a program's meaning in terms of its implementation on a real or virtual machine*
• *Change in the state of the machine (memory, registers, etc.) defines the meaning of the statement*
Operational Semantics

• There are different levels of operational semantics
  – **Big-Step Semantics**: At the highest level, *natural operational semantics* are used to describe the final execution result of a complete program
  – **Small-Step Semantics**: At the lowest level, *structural operational semantics* are used to determine the precise meaning of a single statement
    • How does the statement change the state of a real/virtual machine, such as memory and registers?

Operational Semantics Definition Process

1. Design an appropriate intermediate language. Each construct of the intermediate language must have an obvious and unambiguous meaning
2. Construct a virtual machine (an interpreter) for the intermediate language. The virtual machine can be used to execute either single statements, code segments, or whole programs
An Example

<table>
<thead>
<tr>
<th>C</th>
<th>Operational Semantics</th>
</tr>
</thead>
</table>
| `for (expr1; expr2; expr3) {
  ... break;
}
| `expr1;
  `loop: if expr2 == 0 goto out
  ... break;
  `expr3;
  `goto loop
  `out: ... |

- The virtual computer is supposed to be able to correctly “execute” the instructions and recognize the effects of the “execution”

A Simple Language of Arithmetic Expressions [2]

- **CFG**
  
  \[ e ::= n | e1 + e2 | e1 * e2 \]

- We are curious about:
  - What is the “meaning” of a given ARITH expression?
  - How do we evaluate expression?
Operational Semantics of Arithmetic Expressions [2]

- Specify how expressions should be evaluated
- Defined by cases on the form of expressions
  - $n$ evaluates to $n$
    - $n$ is a normal form, an expression that cannot be reduced further
  - $e_1 + e_2$ evaluates to $n$ if
    - $e_1$ evaluates to $n_1$,
    - $e_2$ evaluates to $n_2$, and
    - $n$ is the sum of $n_1$ and $n_2$

- $e_1 * e_2$ evaluates to $n$ if
  - $e_1$ evaluates to $n_1$,
  - $e_2$ evaluates to $n_2$, and
  - $n$ is the product of $n_1$ and $n_2$
Big-Step Operational Semantics [2]

\[ n \downarrow n \]

\[ e_1 \downarrow n_1 \quad e_2 \downarrow n_2 \quad n \text{ is the sum of } n_1 \text{ and } n_2 \]
\[ e_1 + e_2 \downarrow n \]

\[ e_1 \downarrow n_1 \quad e_2 \downarrow n_2 \quad n \text{ is the product of } n_1 \text{ and } n_2 \]
\[ e_1 \times e_2 \downarrow n \]

Small-Step Operational Semantics [3]

- Describe a single step in the evaluation
- Show intermediate results and how to calculate each result
- Many steps may be needed to get a result
Small-Step Operational Semantics [3]

- The semantic rules tell not only the operator meanings, but also evaluation orders
- E.g., \((1+2) + (3+4) = 3 + (3+4)\)

Key Points of Operational Semantics

- **Advantages**
  - May be simple and intuitive for small examples
  - Good if used informally
  - Useful for implementation
- **Disadvantages**
  - Very complex for large programs
  - Lacks mathematical rigor
Typical Usage of Operational Semantics

- Vienna Definition Language (VDL) used to define PL/I (Wegner 1972)
- Unfortunately, VDL is so complex that it serves no practical purpose

Denotational Semantics

- The most rigorous, widely known method for describing the meaning of programs
- Solely based on recursive function theory
- Originally developed by Scott and Strachey (1970)
Denotational Semantics

• Key Idea
  – Define for each language entity both a mathematical object, and a function that maps instances of that entity onto instances of the mathematical object

• The basic idea
  – There are rigorous ways of manipulating mathematical objects but not programming language constructs

Denotational Semantics

• Difficulty
  – How to create the objects and the mapping functions?

• The method is named *denotational*, because the mathematical objects denote the meaning of their corresponding syntactic entities
Denotational vs. Operational

• Both denotational semantics and operational semantics are defined in terms of state changes in a virtual machine

• In operational semantics, the state changes are defined by coded algorithms in the machine

• In denotational semantics, the state change is defined by rigorous mathematical functions

Program State

• Let the state $s$ of a program be a set of pairs as follows:
  \[ \{<i_1, v_1>, <i_2, v_2>, ..., <i_n, v_n>\} \]
  – Each $i$ is the name of a variable
  – The associated $v$ is the current value of the variable
  – Any $v$ can have the special value undef, indicating that the associated variable is undefined

• Let $\text{VARMAP}$ be a function as follows:
  \[ \text{VARMAP}(i_j, s) = v_j \]
Program State

• Most semantics mapping functions for programs and program constructs map from states to states
• These state changes are used to define the meanings of programs and program constructs
• Some language constructs, such as expressions, are mapped to values, not state changes

An Example

• CFG for binary numbers
  \[
  \text{<bin_num>} \rightarrow \text{'}0\text{'} \\
  \text{<bin_num>} \rightarrow \text{'}1\text{'} \\
  \text{<bin_num>} \rightarrow \text{<bin_num>} \text{'}0\text{'} \\
  \text{<bin_num>} \rightarrow \text{<bin_num>} \text{'}1\text{'}
  \]
• Parse tree of the binary number 110
  \[
  \begin{array}{c}
    \text{<bin_num>} \\
    \quad \text{<bin_num>} \text{'}0\text{'} \\
    \quad \quad \text{<bin_num>} \text{'}1\text{'} \\
  \end{array}
  \]
Example Semantic Rule Design

• **Mathematical objects**
  – Decimal number equivalence for each binary number

• **Functions**
  – Map binary numbers to decimal numbers
  – Rules with terminals as RHS are translated as direct mappings from terminals to mathematical objects
  – Rules with nonterminals as RHS are translated as manipulations on mathematical objects

---

Example Semantic Rules

<table>
<thead>
<tr>
<th>Syntax Rules</th>
<th>Semantic Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;bin_num&gt; -&gt; ‘0’</td>
<td>$M_{\text{bin}}(‘0’)=0$</td>
</tr>
<tr>
<td>&lt;bin_num&gt; -&gt; ‘1’</td>
<td>$M_{\text{bin}}(‘1’)=1$</td>
</tr>
<tr>
<td>&lt;bin_num&gt; -&gt; &lt;bin_num&gt; ‘0’</td>
<td>$M_{\text{bin}}(&lt;\text{bin_num} &gt; ‘0’)=2M_{\text{bin}}(&lt;\text{bin_num} &gt;)$</td>
</tr>
<tr>
<td>&lt;bin_num&gt; -&gt; &lt;bin_num&gt; ‘1’</td>
<td>$M_{\text{bin}}(&lt;\text{bin_num} &gt; ‘1’)=2M_{\text{bin}}(&lt;\text{bin_num} &gt;)+1$</td>
</tr>
</tbody>
</table>
Expressions

- **CFG for expressions**

  \[ <expr> \rightarrow <dec_num> | <var> | <binary_expr> \]

  \[ <binary_expr> \rightarrow <l_expr> <op> <r_expr> \]

  \[ <l_expr> \rightarrow <dec_num> | <var> \]

  \[ <r_expr> \rightarrow <dec_num> | <var> \]

  \[ <op> \rightarrow + | * \]

\[
M_e(<expr>, s) \Delta = \\
\text{case } <expr> \text{ of} \\
\quad <dec_num> \Rightarrow M_{\text{dec}}(<dec_num>) \\
\quad <var> \Rightarrow \text{VARMAP}(<var>, s) \\
\quad <binary_expr> \Rightarrow \\
\quad \quad \text{if } (<binary_expr>.<op> = '+') \text{ then} \\
\quad \quad \quad M_e(<binary_expr>.<l_expr>, s) + \\
\quad \quad \quad M_e(<binary_expr>.<r_expr>, s) \\
\quad \quad \text{else} \\
\quad \quad \quad M_e(<binary_expr>.<l_expr>, s) \times \\
\quad \quad \quad M_e(<binary_expr>.<r_expr>, s)
\]

N. Meng, S. Arthur