

[Next](#)
[Up](#)
[Previous](#)
[Contents](#)

Next: [6 Semantic Analysis](#)
Up: [CSE 5317/4305: Design and](#)
Previous: [4.3 Gen: A Java](#)
[Contents](#)

5 Semantic Actions

Let's consider now how actions are evaluated by different parsers. In recursive descent parsers, actions are pieces of code embedded in the recursive procedures. For the following grammar:

```

E ::= T E'
E' ::= + T E'
      | - T E'
      |
T ::= num

```

we have the following recursive descent parser:

```

int E () { return Eprime(T()); };
int Eprime ( int left ) {
    if (current_token=='+') {
        read_next_token();
        return Eprime(left + T());
    } else if (current_token=='-') {
        read_next_token();
        return Eprime(left - T());
    } else return left;
};
int T () {
    if (current_token=='num') {
        read_next_token();
        return num_value;
    } else error();
};

```

By passing `T()` as input to `Eprime`, we pass the left operand to `Eprime`.

Table-driven predictive parsers use the parse stack to push/pop actions (along with symbols) but they use a separate semantic stack to execute the actions. In that case, the parsing algorithm becomes:

```

push(S);
read_next_token();
repeat
    X = pop();
    if (X is a terminal or '$')
        if (X == current_token)
            read_next_token();
        else error();
    else if (X is an action)
        perform the action;
    else if (M[X,current_token] == "X ::= Y1 Y2 ... Yk")
        { push(Yk);

```

```

    ...
    push(Y1);
  }
  else error();
until X == '$';

```

For example, suppose that `pushV` and `popV` are the functions to manipulate the semantic stack. The following is the grammar of an interpreter that uses the semantic stack to perform additions and subtractions:

```

E ::= T E' $ { print(popV()); }
E' ::= + T { pushV(popV() + popV()); } E'
      | - T { pushV(-popV() + popV()); } E'
      |
T ::= num { pushV(num); }

```

For example, for `1+5-2`, we have the following sequence of actions:

```

pushV(1); pushV(5); pushV(popV()+popV()); pushV(3);
pushV(-popV()+popV()); print(popV());

```

Question: what would happen if we put the action of the second rule at the end of rhs?

In contrast to top-down parsers, bottom-up parsers can only perform an action after a reduction (ie, after the entire rhs of a rule has been processed). Why? because at a particular instance of time we may have a potential for multiple rules for reduction (this is the idea behind itemsets), which means that we may be in the middle of many rules at a time, but later only one rule will actually be used; so, we can't execute an action in the middle of a rule because we may have to undo it later if the rule is not used for reduction. This means that we can only have rules of the form

```

X ::= Y1 ... Yn { action }

```

where the action is always at the end of the rule. This action is evaluated after the rule `X ::= Y1 ... Yn` is reduced. To evaluate actions, in addition to state numbers, the parser pushes values into the parse stack: Both terminals and non-terminals are associated with typed values, which in the CUP parser generator are instances of the `Object` class (or of some subclass of the `Object` class). Since the Java `Object` class is a superclass of all classes, it doesn't carry any additional information, but the subclasses of `Object`, such as the class `Integer`, the class `String`, and the class `Tree` for ASTs, do. The value associated with a terminal is in most cases an `Object`, except for an identifier which is a `String`, for an integer which is an `Integer`, etc. The typical values associated with non-terminals in a compiler are ASTs, lists of ASTs, etc. In CUP, you can retrieve the value of a symbol `s` at the lhs of a rule by using the notation `s:x`, where `@x@` is a variable name that hasn't appeared elsewhere in this rule. The value of the non-terminal defined by a rule is called `RESULT` and should always be assigned a value in the action. For example, if the non-terminal `E` is associated with an integer value (of type `Integer`), then the following rule:

```

E ::= E:n PLUS E:m { : RESULT = n+m; : }

```

retrieves the value, `n`, of the left operand from the parse stack, retrieves the value, `m`, of the right operand from the parse stack, and pushes the value of `RESULT` on the parse stack, which has been set to `n+m` after the reduction of the rule. That is, the elements of the parser stack in CUP are pairs of a state-number (integer) and an `Object`. So when the above rule is reduced, the three top elements of the stack, which form the handle of the reduction, will contain three elements: the state reached when we reduced the rule for `E` to get the left operand,

the state for shifting over `PLUS`, and the state reached when we reduced the rule for `E` to get the right operand (top of stack). Along with these states, there are three Objects: one bound to `n`, one ignored (since the terminal `PLUS` is associated with an empty Object, which is ignored), and one bound to `m` (top of stack). When we reduce by the above rule, we use the GOTO table to find which state to push, we pop the handle (three elements), and we push the pair of this state and the `RESULT` value on the parse stack.

If we want build an AST in CUP, we need to associate each non-terminal symbol with an AST type. For example, if we use the non-terminals `exp` and `expl` for expressions and list of expressions respectively, we can define their types as follows:

```
non terminal Tree      exp;
non terminal Trees    expl;
```

Then the production rules should have actions to build ASTs:

```
exp ::= exp:e1 PLUS exp:e2    {: RESULT = new Node(plus_exp,e1,e2); :}
    | exp:e1 MINUS exp:e2    {: RESULT = new Node(minus_exp,e1,e2); :}
    | id:nm LP expl:e1 RP    {: RESULT = new Node(call_exp,e1.reverse()
                                .cons(new Variable(nm))); :}
    | INT:n                  {: RESULT = new LongLeaf(n.intValue()); :}
    ;
expl ::= expl:e1 COMMA exp:e  {: RESULT = e1.cons(e); :}
    | exp:e                  {: RESULT = nil.cons(e); :}
    ;
```

That is, for integer addition, we build an AST node that corresponds to a binary operation (see the AST in Section 4).

What if we want to put an action in the middle of the rhs of a rule in a bottom-up parser? In that case we use a dummy nonterminal, called a *marker*. For example,

```
X ::= a { action } b
```

is equivalent to

```
X ::= M b
M ::= a { action }
```

This is done automatically by the CUP parser generator (ie, we can actually put actions in the middle of a rhs of a rule and CUP will use the above trick to put it at the end of a rule). There is a danger though that the resulting rules may introduce new shift/reduce or reduce/reduce conflicts.

[Next](#) [Up](#) [Previous](#) [Contents](#)

Next: [6 Semantic Analysis](#) **Up:** [CSE 5317/4305: Design and](#) **Previous:** [4.3 Gen: A Java](#) [Contents](#)
fegaras 2012-01-10